Selective Enumeration

Craig A. Damon

July 2000

CMU-CS-00-151

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA

*Submitted in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy*

Thesis Committee:
Jeannette Wing, co-chair
Daniel Jackson, co-chair
Gary Miller
Somesh Jha
Rance Cleaveland

This research was sponsored in part by the Defense Advanced Research Projects Agency (DARPA), the National Science Foundation (NSF), and the National Institute of Standards and Technology (NIST). The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, NSF, NIST, or the U.S. government.

Keywords

Relational calculus, exhaustive search, model checking, specification checking, constraint satisfaction.

Carnegie Mellon

School of Computer Science

DOCTORAL THESIS
in the field of
COMPUTER SCIENCE

Selective Enumeration

CRAIG A. DAMON

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

ACCEPTED:

Jeannette M. Wing THESIS COMMITTEE CO-CHAIR — May 25, 2000 DATE

Daniel Jackson THESIS COMMITTEE CO-CHAIR — May 25, 2000 DATE

Randal E Bryant DEPARTMENT HEAD — Aug 1, 2000 DATE

APPROVED:

J Moris DEAN — 8/1/00 DATE

Abstract

Selective enumeration is an approach to pruning search trees with the goal of preventing the generation of extraneous paths in the search tree, rather than generating paths that will later be pruned. The reduction in the size of the search tree scales exponentially with both the number of variables and the number of values, making complete coverage of very large (in excess of 1e100 nodes) search trees tractable.

This dissertation demonstrates that selective enumeration enables an analysis of formal specifications of software systems. This analysis discovers counterexamples to user-defined claims about properties of a specification by solving formulae derived the specification. Ladybug is a new tool that incorporates selective enumeration to check software designs. Ladybug's input language is essentially a first-order subset of Z, one of the most broadly used software specification languages.

Ladybug includes implementations of three significant new algorithms to help reduce the search space: bounded generation, domain coloring, and isomorph-eliminating relation generators. Bounded generation improves upon traditional tree pruning techniques by preventing the generation of many subtrees that would subsequently be eliminated. Domain coloring provides an efficient means of building a sound approximation to the automorphism group of an assignment. The isomorph-eliminating generators yield a nearly perfect superset of an isomorph-free set of assignments with minimal cost.

In this thesis, I have applied Ladybug to a suite of software specifications, including both artificial and "real-world" specifications, to quantify the success and failure of Ladybug at analyzing software specifications. I have also used Ladybug as part of a larger case study examining portions of the DoD High Level Architecture specification.

Acknowledgments

My original advisor, Daniel Jackson, developed the basic concept of Nitpick, the predecessor to Ladybug. He also provided much valued advice and support throughout my thesis work. Jeannette Wing, my co-advisor, taught me more about writing and thinking clearly than I thought I would ever know. Somesh Jha served as an advisor, a mentor, and finally a committee member, but I most valued his support as a friend. Gary Miller suffered through months of my not understanding the appropriate group theory. Rance Cleaveland offered support, especially when stranded in the Frankfort airport, and timely advice.

Many friends, especially Rob O'Callahan, Paul Placeway, Belinda Thom, Faye Chang, Darrel Kindred, Rob Deline, and Bob Monroe, helped me in many ways along the torturous path. Elizabeth Bigelow was both a friend and a supporter, convincing Jim "Coach" Tomayko to fund Daniel List of Fonantus, Inc. to develop the original Ladybug user interface. My longtime friend Edith Murphy offered much of her own hard-earned wisdom of the trials of surviving a Ph.D. and taught her husband, Tom Wight, to offer support without asking when it would be done. I also wish to thank the many smiling faces of the Edgewood Eat N' Park morning shift, whose cheerful dispositions and many pots of tea got me through some of the darkest moments.

My parents, Seldra Funk and Craig W. Damon, originally taught me a love of learning and gave me support throughout this process. All my in-laws, particularly Joy Hill and Wally Hill, gave me much needed support. My sister, Sue Havranak, cared for Chris for several years. I especially want to thank my wife Leslie for her many years of love and support and for allowing me to drag he around the country — I don't think she has recovered yet from the shock of me actually applying to graduate school. Finally, I want to thank my pride and joy, my son Chris who grew up with this thesis, for being patient while Daddy finished. Yes, Chris, my book is done.

Table of Contents

Chapter 1

Introduction

Sets, functions, and binary relations offer a convenient, yet rigorous, structure for modeling software systems. Z [Spi92], probably the most widely used formal notation for describing software systems, is based entirely on these constructs. As relational formulae can describe sets, functions, and relations, I use the term *relational specification* to describe any specification built in terms of these constructs.

Other software description notations also draw much of their expressive power from these constructs. Within the database community, the inter-relationships in a database schema are often specified using an entity-relationship diagram [Che76]. Given the name, it should not be surprising that relations are a clear and succinct method of describing entity-relationship diagrams.

More recently, UML [BJR99] has gathered great interest in the object community. Although UML combines several different notations to describe a single object design, many of these notations are built from sets, functions, and relations.

Despite the broad appeal of these constructs, little automated support is available for analyzing relational specifications. Theorem provers [ES94; SM96] can help, but they require enormous manual effort and provide little guidance to help repair faulty specifications. Model checkers [BC+92; CPS93] can analyze system specifications based on other formalisms, but no model checkers are available for relational specifications.

The research described by this dissertation demonstrates that relational specifications are amenable to automated analysis. To support this goal, I develop an approach, numerous techniques, and a tool to analyze relational specifications. Selective enumeration is an approach that reduces the size of the search tree used in the analysis. Bounded generation and isomorph-eliminating generators are the two most noteworthy techniques developed to support this thesis. Ladybug is a new tool that implements these techniques and has been used to discover flaws in real-world relational specifications.

1.1 Alloc — An Example

To help ground these ideas, this section introduces a very simple relational specification that describes a heap allocation system, such as malloc, in very general terms. I will use this example to illustrate points throughout the remainder of this dissertation.

The specification is written in NP [JD96a], a relational specification language that is roughly a subset of Z. NP is limited to first-order objects, so, for example, NP does not allow functions of functions. Figure 1.1 contains the NP specification for the heap allocation system.

```
[Addr, Data]

Heap =
[
 usage : Addr -> Data
 used : set Addr

|

 /* all currently mapped addresses are used */
 used = dom usage
]
Alloc(addr : Addr) =
[
 Heap

|

 /* Allocating a new address does not change the current allocation */
 used <: usage' = usage

 /* But addr is now mapped (to some unknown data element) */
 used' = used U {addr}
 ]
uniqueAddrAlloc::
[
 Heap
 newAddr : Addr

|

 /* A newly allocated address should not have been in use */
 Alloc(newAddr) => newAddr not in used
]
```

Figure 1.1. A trivial NP specification describing a heap allocation system. Addr and Data are the given types. Heap describes the basic structure being manipulated, Alloc describes an allocation operation, and uniqueAddrAlloc is a claim about the specification.

The first line of the example introduces the two given types used in this specification, Addr and Data. A given type is a set of elements, with each element being uninterpreted and having no internal structure. Every element is contained in exactly one given type. All variables and expressions in NP are typed, indicating that they refer to one of three kinds of values. A variable or expression may refer to (1) an element of a given type, (2) a set of elements of a single given type, or (3) a relation that maps elements of a given type (the domain) to elements of a given type (the range). Relations can be restricted to be functions, injections, or bijections and to be total or surjective relations.

When using NP, specifiers describe their system using a collection of schemas, which allow a simple structuring and composition of individual pieces of the specification, similar to the mechanism provided by Z. Two independent characteristics jointly classify schemas in NP. A schema is either a *description*, which defines the system being specified, or a *claim*, which makes assertions about the system being specified. A schema, whether a description or a claim, refers either to a single state or to a transition between two states. A transitional schema is called an operation and describes both a pre-state and a post-state. The specification given in Figure 1.1 contains examples of three of the four possible combinations of these characteristics, as explained in the following paragraphs.

All schemas have the same basic structure. The body of the schema comes after the name of the schema and is enclosed in square brackets ([]). A single vertical bar (|) divides the body into two sections. The first section defines the variables used in the schema, whereas the second section gives a collection of relational formulae that must be satisfied in any system described by this specification.

In the example given in Figure 1.1, Heap is a descriptive schema that describes the basic structure of a heap. Heap introduces two variables, usage and used. The variable usage denotes a function mapping addresses (elements of Addr) to their data (elements of Data). The other variable, used, denotes a set that contains all the addresses currently in use. Heap also defines a single formula that describes a relationship that must hold in all valid heaps: the set of addresses in use is exactly the set of addresses currently mapped, that is, the domain of the function usage.

Alloc is an operation that describes the change in a heap when a new piece of memory is allocated. As Alloc refers to Heap in its declaration section, Alloc inherits all the variables defined by Heap. Within Alloc, the pre-state is referenced using the simple variable names, whereas the post-state is referenced using primed variables, such as usage'. Operations are indicated by the presence of a (possibly empty) parameter list. The parameter list for Alloc defines a single parameter, addr, which is the newly allocated address.

Alloc contains two formulae. The first (used <: usage' = usage)[1] guarantees that the allocation does not change any existing mappings. The second formula (used' = used U addr) indicates that the newly allocated address is now considered to be in use (in addition to any addresses already in use).

The third schema, uniqueAddrAlloc, is a claim that asserts that the newly allocated address is not in use prior to the allocation.

1.2 Generate-and-Test Searching

A method for solving relational formulae must lie at the core of any automated tool for analyzing relational specifications. The simplest approach is a generate-and-test search. A generate-and-test search generates all possible mappings of variables to values, called *assignments*, for a particular formula. The search tests each generated assignment against that formula. The result of the search is a set of satisfying assignments, that is, assignments that give a true interpretation to the formula.

An exhaustive-enumeration search is the simplest generate-and-test search that is sound. Exhaustive enumeration generates all possible assignments in a search tree, with each level of the search tree corresponding to a distinct variable in the formula. Testing a single assignment against a formula is straightforward, requiring only an implementation of the standard boolean, set, and relational operations.

However, using exhaustive-enumeration search as a solver presents two limitations. By its nature, a generate-and-test search will consider only some finite subset of the (generally infinite) possible assignment space. Although this limitation prevents a generate-and-test search from being a true verifier for infinite problems, it does not remove all practical applications. As I believe that many, if not most, errors in specifications can be demonstrated using only a small subset of

[1] The <: operator is the domain restriction operator. The result of this expression is a relation that includes all of the pairs in the relation given as the second argument whose first element is contained in the set given as the first argument. For the formal definition of this and other operators, refer to the definitions beginning on page 16.

the entire assignment space, a generate-and-test search can be the basis of a practical specification analysis tool.

The second limitation is the time required to generate and test the complete set of assignments. This limitation has far more significant practical implications. For the trivial alloc example with three addresses and three data elements, the number of full assignments is a tractable 786,432. Raising the number of addresses and data elements to five apiece raises the number of full assignments to consider to a more problematic 10^{11}. A slightly more complicated specification (such as finder [JD95]) with only five underlying objects increases the total number of assignments required to generate and test to more than 10^{27}.

Generating all 10^{27} assignments is clearly inconceivable, rendering exhaustive enumeration impractical. Fortunately, the vast majority of these assignments are in some sense "duplicates" of other assignments. One assignment may be isomorphic to another assignment. Two assignments may share some common partial assignment, which itself determines the interpretation of the formula. Regardless of the nature of the duplication, generating only one assignment from each set of duplicate assignments is sufficient.

Selective enumeration is a generate-and-test search method that prevents the generation (and therefore the testing) of most duplicates. By preventing the generation of these duplicates, selective enumeration is effective in solving many interesting relational formulae.

1.3 Reducing the Search

Attempting to validate claims such as uniqueAddrAlloc is a common analysis of NP specifications. A claim is valid if there are no assignments that satisfy the negation of the claim. Ladybug, the tool that I have implemented to analyze relational specifications, validates claims (within user specified finite bounds) using selective enumeration to solve the negation of the claim. The satisfying assignments for the negation of the claim are counterexamples of the claim, which can be presented to the user. For the claim uniqueAddrAlloc, a counterexample must satisfy the schemas Heap and Alloc but violate the consequent newAddr not in used.

With 3 addresses and 3 data elements, there are 786,432 possible full assignments of values to variables. Most of these assignments fail to satisfy the requirements of a valid invocation of the alloc operation. For example, if the elements of Addr are { a_0, a_1, a_2 } and the elements of Data are { a_0, a_1, a_2 }, the assignment

$$\begin{array}{l} \mathrm{newAddr} = a_1 \\ \mathrm{usage} = \{\, a_0 \mapsto d_1,\ a_2 \mapsto d_0 \,\} \\ \mathrm{used} = \{\, a_0 \,\} \\ \mathrm{usage}' = \{\, a_0 \mapsto d_2 \,\} \\ \mathrm{used}' = \{\, a_1,\ a_2 \,\} \end{array}$$

fails to satisfy the constraint used = dom usage, among others. Of the 288 assignments that satisfy the requirements of a valid invocation of alloc, 144 are counterexamples to the claim uniqueAddrAlloc, including

$$\begin{array}{l} \mathrm{newAddr} = a_0 \\ \mathrm{usage} = \{\, a_0 \mapsto d_0 \,\} \\ \mathrm{used} = \{\, a_0 \,\} \\ \mathrm{usage}' = \{\, a_0 \mapsto d_0 \,\} \\ \mathrm{used}' = \{\, a_0 \,\} \end{array}$$

Ladybug prunes most of the assignments as duplicates. A complete search by Ladybug considers only 18 full assignments, discovering 7 distinct counterexamples. Each of the remaining 137 counterexamples duplicates one of the 7 counterexamples that Ladybug discovers.

To perform this reduction, Ladybug recognizes and exploits two kinds of duplications: partial-assignment duplicates and isomorph duplicates. Two assignments are partial-assignment duplicates if they share a common mapping of values for a subset of the variables (called a partial assignment) and that partial assignment itself determines the value of the formula. Two assignments are isomorph duplicates if one is isomorphic to the other.

In many specifications, the values of some variables are defined constructively, that is, their value is constrained to be equal to a function of the values of the other variables. In the example, Heap defines the formula used = dom usage. Therefore, the value of used is constrained to be equal to the domain of the value of usage in any counterexample to uniqueAddrAlloc.

The simplest way to exploit partial assignment duplicates is with *derived variable elimination* [JD95]. Variables with a constructive definition are the most common example of derived variables. Given the bindings for the other variables, the search can directly construct the value of a derived variable, rather than generating many possible values and testing each one. Assuming that usage is bound prior to used being generated, Ladybug computes the value of used.

Selective enumeration requires the imposition of a variable ordering. Although any ordering is legal for selective enumeration, some orderings yield a much greater reduction in the number of assignments generated than is yielded by other orderings. I discuss the choice of orderings in Chapter 5.

Because the constraint newAddr not in used must be violated, the value of newAddr must be an element of the value of used in any counterexample to uniqueAddrAlloc. Although this constraint does not limit the possible values of newAddr to a single value, the constraint can be used to limit the values actually generated during the search. *Bounded generation* uses constraints from the formula to limit the values generated. Assuming that used is bound before the value of newAddr is generated, bounded generation will generate each element in the set that is the value of used, instead of each value in the given type Addr.

A second opportunity for bounded generation exists in uniqueAddrAlloc. The first formula in Alloc, used <: usage' = usage, must be true for any counterexample to uniqueAddrAlloc. To simplify the implementation, bounded generation does not directly take advantage of this constraint; instead, bounded generation uses a weaker constraint, usage <= usage', that is implied by the original formula. This weaker constraint allows bounded generation to limit both the domain and range of any value generated for usage to be subsets of the domain and range of the value of usage'.

Derived variable analysis and bounded generation cannot fully exploit all formulae within a specification. If these formulae do not depend on all the variables, they still present an opportunity for reducing the assignments to be generated. *Short circuiting* [DJ96] does not reduce the number of values generated for any variables involved in the formula, as would bounded generation. Instead, short circuiting prevents generation of values for any subsequent variables when the partial assignment cannot satisfy the formula.

An example of short circuiting can be found for the constraint on usage and usage' that initiated the second bounded generation example. Although bounded generation will guarantee that dom usage <= dom usage' and ran usage <= ran usage', this constraint does not guarantee that usage <= usage'. Once values have been bound to both usage and usage', short circuiting evaluates the constraint usage <= usage' for the resulting partial assignment. Short circuiting will terminate the current path of the search for any partial assignments not satisfying the constraint. Similarly, once

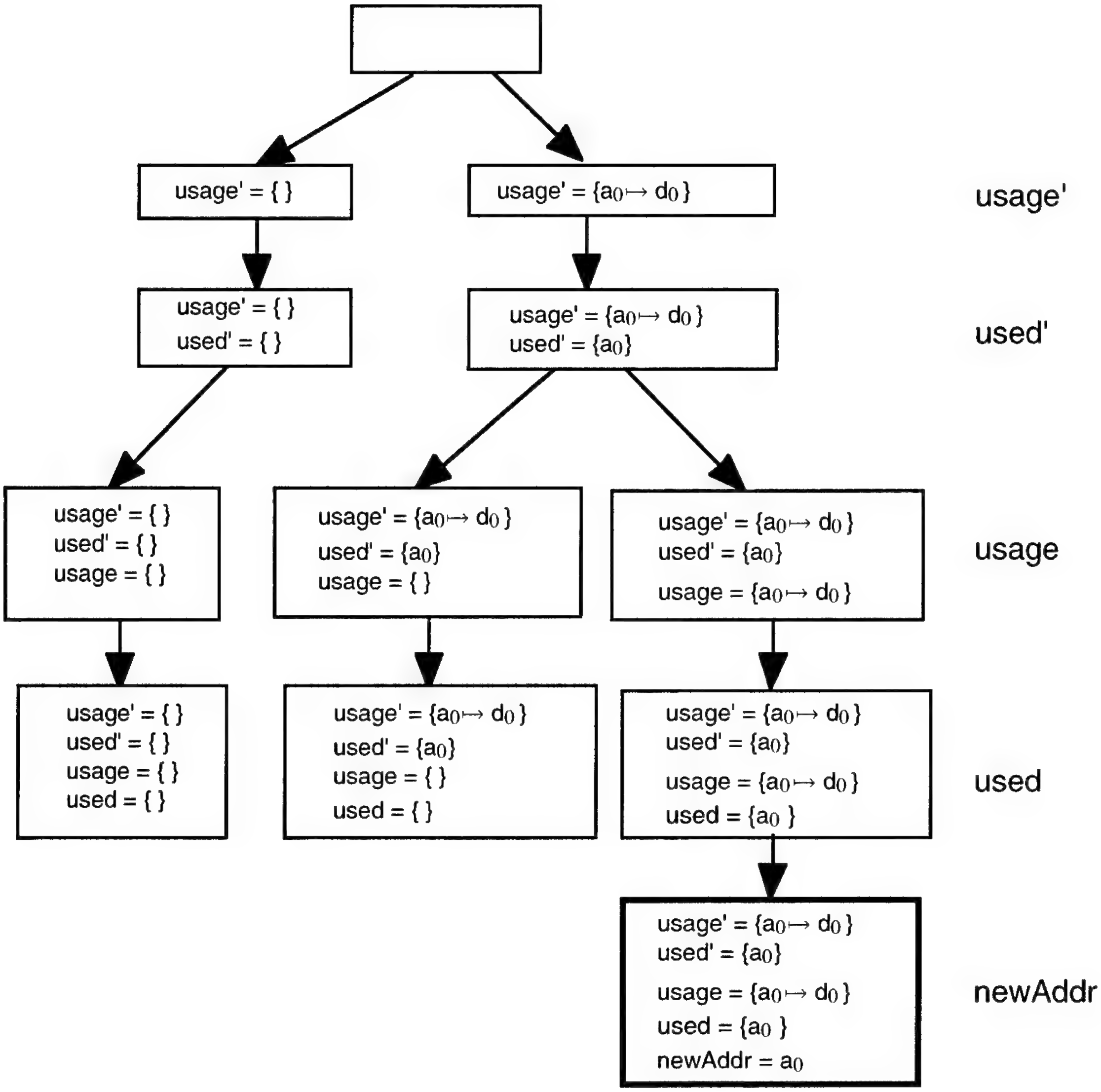


Figure 1.2. The search tree for finding a counterexample to the claim uniqueAddrAlloc. The variables used' and used are derived; Ladybug can compute their values directly from the earlier assignments. Bounded generation limits the domain and range of the values generated for usage to a subset of the domain and range used in usage'. Similarly, bounded generation limits the values considered for newAddr to the elements in the value of used. The first two paths down the search tree result in used being empty, leaving no possible values for newAddr. The heavier box illustrates the first counterexample discovered.

usage, usage', and used have been generated, short circuiting will check the full constraint, used <: usage' = usage. By utilizing all three techniques, selective enumeration can eliminate all partial assignment duplicates available with the selected ordering.

Ladybug uses these three techniques, bounded generation, derived variables, and short circuiting, to remove partial assignment duplicates. The second form of duplication exploited by Ladybug is called isomorph duplication. Because each element in a given type is unstructured, exchanging a pair of elements throughout an assignment does not change the interpretation of the formula for that assignment. Isomorph elimination [JJD96;JJD98] prevents the generation of most[2] values that are isomorphic to other values already generated.

As an example of isomorph elimination, consider the values generated for usage'. If Addr and Data are limited to three elements apiece, it is necessary to generate 64 ($(\#\text{range}+1)^{\#\text{domain}}$) values for the partial function usage' without isomorph elimination. With isomorph elimination, on the other hand, only the following seven values need to be generated:

$\text{usage}' = \varnothing$
$\text{usage}' = \{ a_0 \mapsto d_0 \}$
$\text{usage}' = \{ a_0 \mapsto d_0, a_1 \mapsto d_1 \}$
$\text{usage}' = \{ a_0 \mapsto d_0, a_1 \mapsto d_0 \}$
$\text{usage}' = \{ a_0 \mapsto d_0, a_1 \mapsto d_1, a_2 \mapsto d_2 \}$
$\text{usage}' = \{ a_0 \mapsto d_0, a_1 \mapsto d_0, a_2 \mapsto d_1 \}$
$\text{usage}' = \{ a_0 \mapsto d_0, a_1 \mapsto d_0, a_2 \mapsto d_0 \}$

The result of this reduced search is illustrated in Figure 1.2 and Figure 1.3. Figure 1.2 demonstrates the search until the first counterexample is found. When the number of elements in Addr and Data are limited to three apiece, derived variable elimination and bounded generation reduce the search to find the first counterexample from 13,851 assignments to just 3. Figure 1.3 expands the tree for one more value of usage', exhibiting the further advantages of short circuiting and isomorph elimination. Selective enumeration includes two general categories of reductions: cases where two or more assignments are known to be both satisfying or both not satisfying for a formula, without explicitly testing them, and cases where two or more assignments are known to give the same unknown interpretation to the formula. Bounded generation, short circuiting, and derived variables all exploit the former case, and isomorph elimination exploits the latter case.

Ladybug also incorporates one approach that is compatible with selective enumeration, but is not a form of selective enumeration: solving one or more related formulae that are somehow simpler to solve and have equivalent satisfying assignments. Normalizing the formula and type rewriting are the two examples of this approach that Ladybug employs.

1.4 The Thesis

In this dissertation, I show that selective enumeration is a useful approach to reduce the cost of a search. In particular, I demonstrate that selective enumeration reduces, to a practical level, the cost of otherwise infeasible searches that solve relational formulae derived from Z-like specifications.

My thesis offers three major contributions:

1) the selective enumeration framework for understanding and implementing search tree pruning techniques;

2) new techniques and algorithms for pruning search trees defined by relational formulae;

3) the Ladybug checker, which implements these techniques and supports the first practical analysis of many Z-like specifications;

[2] The implementation of isomorph elimination does not consider all possible isomorphisms. In particular, only products of selected single permutations are considered.

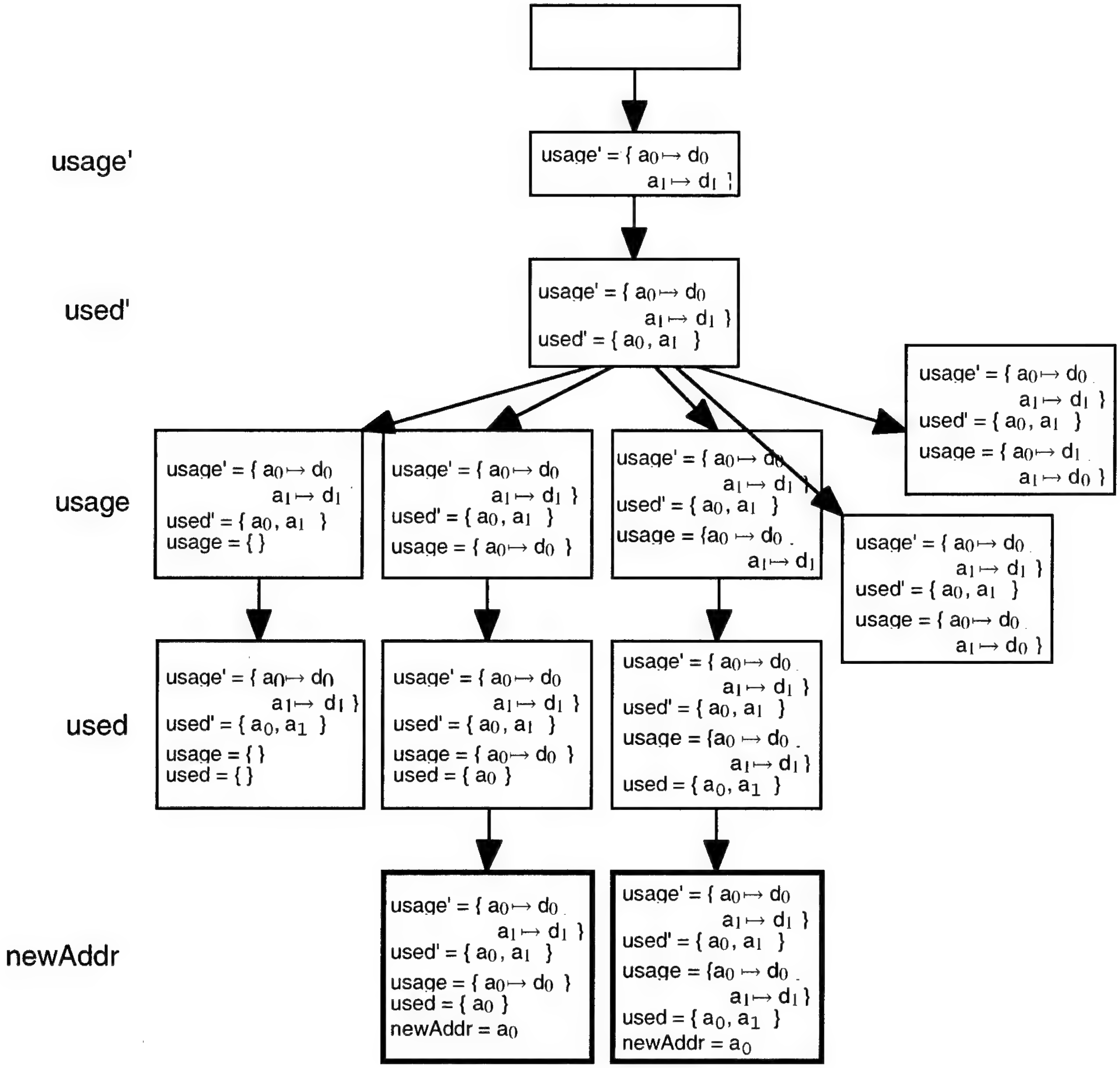

usage'
used'
usage
used
newAddr
usage' = { a0↦ d0 a1↦ d1 }
usage' = { a0↦ d0 a1↦ d1 } used' = { a0, a1 }
usage' = { a0↦ d0 a1↦ d1 used' = { a0, a1 } usage = { }
usage' = { a0↦ d0 a1↦ d1 } used' = { a0, a1 } usage = { a0↦ d0 }
usage' = { a0↦ d0 a1↦ d1 } used' = { a0, a1 } usage = {a0 ↦ d0 . a1↦ d1
usage' = { a0↦ d0 a1↦ d1 } used' = { a0, a1 } usage = { a0↦ d1 . a1↦ d0 }
usage' = { a0↦ d0 a1↦ d1 } used' = { a0, a1 } usage = { a0↦ d0 . a1↦ d0 }
usage' = { a0↦ d0 a1↦ d1 } used' = { a0, a1 } usage = { } used = { }
usage' = { a0↦ d0 a1↦ d1 } used' = { a0, a1 } usage = { a0↦ d0 } used = { a0 }
usage' = { a0↦ d0 a1↦ d1 } used' = { a0, a1 } usage = {a0 ↦ d0 . a1↦ d1 } used = { a0, a1 }
usage' = { a0↦ d0 a1↦ d1 } used' = { a0, a1 } usage = { a0↦ d0 } used = { a0 } newAddr = a0
usage' = { a0↦ d0 a1↦ d1 } used' = { a0, a1 } usage = {a0 ↦ d0 . a1↦ d1 } used = { a0, a1 } newAddr = a0

Figure 1.3. Continuation of the search tree from Figure 1.2. The heavier boxes indicate satisfying assignments. Isomorph elimination generated $\{a_0 \mapsto d_0, a_1 \mapsto d_1\}$ as the next value for usage' because all other single-edge values are isomorphic to the one already generated in Figure 1.2 ($\{a_0 \mapsto d_0\}$). Short circuiting truncates the search for the rightmost two values generated for usage, as these partial assignments do not satisfy the requirements of Alloc. In particular, these partial assignments do not satisfy the formula usage <= usage', which is derived from used <: usage' = usage.

1.5 Related Work

This section provides a broad overview of how other work relates to the work described in this dissertation. Sections at the end of many subsequent chapters describe how the specific techniques relate to other similar techniques. The general related work falls into two basic groups: efforts to validate properties of specifications and techniques to reduce the cost of a search, regardless of domain.

Validating properties of specifications is a central issue in formal methods [CW+96]. The formal methods community has developed three approaches to check specifications: theorem provers, model checkers, and relational formula solvers.

Automated theorem provers [ES94; SM96] validate a property by developing a proof of the validity, much as a human would. This approach offers two advantages over the other approaches. If the property is valid, the final result is a proof of that validity, which can be cross checked by humans or other tools. This cross-checking can provide a significantly greater degree of confidence in the validated properties than is provided by counter-example generators. Furthermore, the validity is not limited by the finite bounds generally required in other approaches.

Theorem provers have their downside as well. Generating a proof for the validity of a nontrivial property can be exceedingly time consuming. Furthermore, theorem provers offer little guidance when the property is invalid.

The other approaches offer their greatest support when the property is invalid; these approaches produce a concrete counterexample for invalid properties. They also typically produce these counterexamples quickly and with minimal human intervention.

Model checking [BC+92; CPS93] is a broadly used approach that exhibits these advantages. Although several different techniques have been developed to implement model checking, all the techniques solve the same problem. The specification to be analyzed describes the graph of all possible states for a system by describing one or more starting states and a transition relation that extends these states into all reachable states. Properties to be validated state that a set of paths (or trees), which may be cyclic or even infinite, do or do not exist in the graph of states. The specifications analyzed by model checkers must describe individual states in the path in very concrete ways; these specifications can describe general properties such as cycles within the data structures clumsily, if at all.

This pairing of powerful expressive power for describing paths with limited expressive power for describing individual states is the inverse of the situation for the relational formula solvers, including Ladybug. Relational formulae can describe paths only by enumerating exact sequences of operations, but they provide more descriptive power for individual states.

The other two relational formulae solvers, the BDD version of Nitpick [DJJ96] and Alcoa [Jac98], both translate a relational formula into a boolean formula and then apply existing boolean satisfiability systems to find solutions to the boolean formula. Unlike Ladybug, where the effort is to reduce the size of the search space by removing duplicates, the effort in these tools is finding an efficient mechanism to translate the relational formula into an appropriate boolean formula. Whereas Ladybug uses only a small amount of memory that grows approximately linearly with the size of the specification, the BDD version of Nitpick is exponential in size as well as time.

The other area of work that is broadly relevant to this thesis is the extensive development of search techniques. Historically, many different forms of search have been considered; only the work that can be expressed as searching for a solution to a formula are relevant to this dissertation. Four other communities are actively researching this form of search: planning, constraint satisfaction, boolean satisfaction, and model finding.

In general, planners [FN71;BW94; BF97] look to bind one or more actions to each time step, with the complete sequence of actions satisfying the requirements of the goal. In terms of selective enumeration, the variables are equivalent to the time steps, the universe of elements are the possible actions, and the values are individual actions, sets of actions, or relations mapping actors to actions, depending on the exact framework. Planners incorporate techniques that are domain specific as well as general search reduction techniques similar to selective enumeration. The general techniques are described in the related work sections in the relevant chapters. No effort in planning has fully formalized or generalized these techniques. Selective enumeration can cleanly describe (and regularize) a large portion of the work done by the planning community.

Constraint satisfaction [Kum92] more obviously solves a problem that fits the selective enumeration framework. Constraint satisfaction binds values to variables that satisfy the constraints described in the problem. Haralick and Elliot [HE80] define a statistical model that describes the effectiveness of simplified versions of the various search reduction techniques commonly employed, including generate and test, backtracking, forward checking, and look ahead. Van Hentenryck [VHe89] provides a formal framework that cleanly describes these techniques. Dechter and Frost [DF98] provide a more refined formal model to distinguish the variations of backtracking and forward checking. However, all these models describe only a common subset of all possible constraint satisfaction problems; each individual constraint must involve no more than two variables. Despite the common use of phrases such as "pruning duplicates from the search tree" in the explanatory text, none of these formal models describe duplication. As a result, these models ignore isomorph elimination, instead considering only a subset of what I term partial-assignment reductions.

Constraint satisfaction algorithms also place restrictions on the problems they can solve. Waltz's classic shape recognition algorithm [Wal75], as well as many later CSP (constraint satisfaction problem) algorithms, requires a complete enumeration of the possible values for each variable. This approach is not feasible for relational formulae, where the number of values for a single variable may number in the millions. Mackworth [Ma77] generalized Waltz's algorithm into arc-consistency, which supports only binary constraints. Other tools, such as the one from Lee and Plaisted [LP94], require the formula to be expressed as Horn clauses, which is less expressive than the relational language.

Some constraint problems, such as job shop scheduling [Fox83], are similar in complexity and scale of the problems to those handled by Ladybug. Although many interesting techniques have been developed, no general formal framework exists that can describe those techniques.

Although boolean satisfaction cleanly and exactly fits the description of selective enumeration and has been amenable to practical solutions [DP60;Bry92;SLM92], the two most powerful techniques incorporated in Ladybug, bounded generation and isomorph elimination, are not applicable to boolean formulae. Bounded generation reduces the number of values by removing some of the values from the universe of values based on the structure of this or previous values. Boolean values are obviously atomic and unstructured and no portion of them can be removed. Similarly, no symmetry exists with boolean values. Symmetries may exist in the variables; exploiting this symmetry, however, would require a completely different approach from the one described here.

The problem solved by the model finding community, such as Finder [Sla94] and SEM [ZZ95a], is similar to the problem solved by Ladybug. Like Ladybug, model generators search for solutions to relation formulae. Significant differences exist in both the formula language and the target problems. Whereas the relational formula language supports only binary relations, the model finding community considers general n-ary relations. Ladybug is tuned to handle formulae with several variables using small scopes; the model finders expect to handle problems with only one or two variables with larger scopes. These differences impact both the techniques and the implementation of the tools. These differences are discussed in the related works sections of the upcoming chapters. Although the model finding tools may exploit limited forms of both isomorph elimination and partial assignment reduction, none provide a framework for considering their interactions.

1.6 Structure of The Dissertation

Chapter 2 formally defines selective enumeration and provides the framework for describing each of the techniques. It defines the basic elements of selective enumeration including search, duplication, and generators, as well as desirable properties such as soundness, reduction, and efficiency. The definitions are mostly applicable to solving a broad range of problems, including relation formulae, but a few of the definitions deal specifically with the issues arising when solving relational formulae. For the reader's convenience, Appendix A repeats the key definitions from the dissertation.

Chapter 3 describes the partial-assignment techniques, beginning with a formal definition of each technique. One section details the working of bounded generation. The chapter follows with a discussion of estimating the possible benefits of partial-assignment duplication. The chapter concludes with a review of similar techniques applied to search optimizations.

Chapter 4 describes isomorph elimination. It begins with the formal definitions necessary to describe the isomorph duplication and generators precisely. The chapter also addresses the interactions between techniques utilizing the two duplications. Finally, the chapter includes an overview of other related work.

Chapter 5 addresses implementation issues involved in building Ladybug. It provides an overview of the architecture of the tool, an overview of isomorph elimination, and details about some of the specific mechanisms used to support partial assignment reductions, such as the heuristic for choosing a variable ordering and the process for discovering facts about the formula.

Chapter 6 provides empirical evidence for the success of selective enumeration as implemented in Ladybug. It describes a benchmark suite, consisting of twenty claims and operations drawn from eleven specifications. The chapter provides measurements of both the time and the number of cases and values required by Ladybug to analyze these specifications. The measurements demonstrate both the overall effectiveness of Ladybug, as well as the effectiveness of each constituent technique.

Although Chapter 6 provides a description of each specification in the benchmark suite and Appendix B contains the complete text of each specification, other chapters sometimes refer to these specifications and a brief summary is worthwhile here. The specifications underlying the mobileIP specification, the two HLA specifications, and the coda specification are "real world" specifications, in that they were written as part of an attempt to discover problems in real systems. The digicash specification and the faa specification are smaller analyses of real systems. The styles specification, the math specification, the phone specification, and the finder specification, along with the previously introduced alloc specification, are artificial, having been generated more to investigate the properties of the tool than the properties of real systems.

Chapter 7 describes an analysis of the HLA system specification [DOD97]. This case study focuses on two aspects of the entire system: ownership properties and bridge federates. After I formalized appropriate portions of the specification (into NP), Ladybug checked several claims about the system. I discovered flaws in the original specification during both the formalization and the checking. While providing a "real-world" context for analyzing relational specifications, this chapter focuses on the performance of Ladybug during the checking.

Chapter 8 concludes the dissertation. It shows how selective enumeration can be used to solve other problems and places selective enumeration in its larger context.

Chapter 2

Basic Definitions

This chapter develops the terminology required to define selective enumeration precisely. In the next two chapters, I use this terminology to define each selective enumeration technique used by Ladybug to reduce the size of the search.

In Chapter 1, I demonstrated how Ladybug, using selective enumeration, discovers counterexamples of a claim about a system specified in NP. The user of Ladybug provides a formula that describes the claim and a scope that defines the set of values to be considered in the search. I call this pairing of a formula and a set of values the *problem* to be solved.

However, solving relational formulae is only one of many possible domains for selective enumeration. A *problem domain* for selective enumeration consists of two elements: (1) a language definition and (2) a universe of values. A problem is constrained by its problem domain: the formula must be an element of the language and the set of values for the search must be a subset of the universe.

Although in this thesis I will focus on using selective enumeration to solve relational formulae, most of the definitions in this chapter describe selective enumeration generally, across all problem domains. Definitions that apply only to the relational problem domain are starred. Supporting other problem domains requires developing equivalent domain specific definitions.

The first two sections of this chapter formally develop the relational problem domain, while providing an overview of the requirements for defining a problem domain for selective enumeration. The next three sections define the key concepts of selective enumeration for any domain: search, duplications, and generators. The final section develops a framework for comparing selective-enumeration techniques.

2.1 Values and Variables

One of the two elements in the definition of a problem domain is the underlying set of values, called *Value*. Each problem domain must define a finite set of values over which the variables vary.

For the relational problem domain, the set of values is derived from a finite universe of unstructured and uninterpretted elements, which I call *U*. To simplify the presentation, I ignore the type distinctions between elements for most of this dissertation. Therefore, for the simple alloc example described in Chapter 1, *U* includes both the Addr and Data sets.

***Definition 2.1** (*U - for Relational Problem Domain*)

The universe of atomic elements, *U*, is a finite set of unstructured and uninterpreted elements.

For the relational problem domain, *Value* contains three kinds of values, all constructed from *U*. The kinds of values are (1) atomic elements of *U*, (2) sets of atomic elements, and (3) binary relations on the atomic elements.

***Definition 2.2** (*Value - for Relational Problem Domain*)

$Value_{\mathrm{scalar}} = U$
$Value_{\mathrm{set}} = \mathbf{P}\ U$
$Value_{\mathrm{rel}} = \mathbf{P}\ (U \times U)$
$Value = Value_{\mathrm{scalar}} \cup Value_{\mathrm{set}} \cup Value_{\mathrm{rel}}$

The search binds these values to variables. The variables of interest are the variables of the formula to be solved.

Definition 2.3 (*Variable*)

The set *Variable* includes all variables used in the formula being solved.

I define *N* to be the number of variables in the formula for a problem.

Definition 2.4 (*N*)

$N = |\ Variable\ |$

For many problem domains, including the relational problem domain, the variables are typed; any given variable can only be bound to a subset of *Value*. I describe this requirement with a typing function.

Definition 2.5 (*Typing*)

The function $Typing : Variable \longrightarrow \mathbf{P}\ Value$ describes the subset of values that may be bound to each variable.

For the relational formulae problem domain, I partition the complete collection of variables into three sets based on the kind of value they can denote: Var_{scalar}, Var_{set}, and Var_{rel}.

***Definition 2.6** (*Variable - for Relation Problem Domain*)

Var_{scalar}, Var_{set}, and Var_{rel} partition the set *Variable*, with the constraints that

$\forall v \in Var_{\mathrm{scalar}} \cdot Typing\ (v) \subseteq Var_{\mathrm{scalar}}$
$\forall v \in Var_{\mathrm{set}} \cdot Typing\ (v) \subseteq Var_{\mathrm{set}}$
$\forall v \in Var_{\mathrm{rel}} \cdot Typing\ (v) \subseteq Var_{\mathrm{rel}} Value_{\mathrm{scalar}})$

For the claim uniqueAddrAlloc from Figure 1.1, *N* is 5 and the variables are

$Var_{\mathrm{scalar}} = \{\ \mathrm{newAddr}\ \}$
$Var_{\mathrm{set}} = \{\ \mathrm{used}, \mathrm{used}'\ \}$
$Var_{\mathrm{rel}} = \{\ \mathrm{usage}, \mathrm{usage}'\ \}$

An *assignment* is a mapping from variables to appropriate values. An assignment can be a full assignment, mapping all variables to appropriate values, or it can be a partial assignment, mapping only a subset of the variables to values.

Definition 2.7 ***(A)***

The set of assignments $A : \mathbb{P}(\mathit{Variable} \twoheadrightarrow \mathit{Value})$ is defined as

$$\{\ \mathrm{a} : \mathit{Variable} \twoheadrightarrow \mathit{Value}\ \mid\ \forall(\mathrm{var},\mathrm{value}) \in \mathrm{a}.\ \mathrm{value} \in \mathit{Typing}(\mathrm{var})\ \}$$

2.2 Relational Language

The *Value* set is one half of the definition of a selective-enumeration problem domain. The other half is the definition of the formula language. Intuitively, any "well-behaved" formula language can be used, i.e. a language whose interpretation is determined by the binding of the variables used in a formula. In this section, I define a formula language for the relational problem domain, which is used for the examples throughout this thesis. I also note specific requirements for a language to be supported by selective enumeration.

The relational problem domain could use any one of the many traditional relational formula languages. The formula language I have chosen is based on the language NP [JD96a], which, in turn, is based on the specification language Z [Spi92]. The formula language is a simplification of NP in two significant ways. The schema constructs, which simplify both the writing and reading of a specification for humans, add no semantic value and are excised from the formula language. Many operators in NP are semantically equivalent to the combination of other operators and have been dropped from the formula language.[1] To simplify the presentation of the thesis, I omit a number of operators which cannot be expressed from the remaining operators. These operators, including function application and the universal relation, were not required by any of the examples and introduce no new complications.

The foundation of the formula language for any problem domain is the terms. Terms describe the ways that values can be constructed using the language. As with other items in the relational problem domain, I divide terms into three categories: $\mathit{Term}_{\text{scalar}}$, $\mathit{Term}_{\text{set}}$, and $\mathit{Term}_{\text{rel}}$.

Definition 2.8** ***(Term)

$$\mathit{Term} = \mathit{Term}_{\text{scalar}} \cup \mathit{Term}_{\text{set}} \cup \mathit{Term}_{\text{rel}}$$

$$\begin{aligned}
\mathit{Term}_{\text{scalar}} \quad &::= \quad \mathit{Var}_{\text{scalar}} \\
\mathit{Term}_{\text{set}} \quad &::= \quad \mathit{Var}_{\text{set}} \,\big|\, \{\ \mathit{Term}_{\text{scalar}}\ \} \,\big|\, \{\,\} \,\big|\, \mathsf{Un} \,\big| \\
&\qquad (\ \mathit{Term}_{\text{set}}\ \mathsf{U}\ \mathit{Term}_{\text{set}}\ ) \,\big|\, (\ \mathit{Term}_{\text{set}}\ \&\ \mathit{Term}_{\text{set}}\ ) \,\big| \\
&\qquad (\ \mathit{Term}_{\text{set}} \setminus \mathit{Term}_{\text{set}}\ ) \,\big|\, \mathit{Term}_{\text{rel}}(\mathit{Term}_{\text{set}}) \,\big| \\
&\qquad \mathsf{dom}\ \mathit{Term}_{\text{rel}} \,\big|\, \mathsf{ran}\ \mathit{Term}_{\text{rel}} \\
\mathit{Term}_{\text{rel}} \quad &::= \quad \mathit{Var}_{\text{rel}} \,\big|\, (\ \mathit{Term}_{\text{set}} <: \mathit{Term}_{\text{rel}}) \,\big|\, \mathsf{Id} \,\big| \\
&\qquad (\ \mathit{Term}_{\text{rel}}\ \mathsf{U}\ \mathit{Term}_{\text{rel}}\ ) \,\big|\, (\ \mathit{Term}_{\text{rel}}\ \&\ \mathit{Term}_{\text{rel}}\ ) \,\big| \\
&\qquad (\ \mathit{Term}_{\text{rel}} \setminus \mathit{Term}_{\text{rel}}\ ) \,\big|\, (\ \mathit{Term}_{\text{rel}}\ ;\ \mathit{Term}_{\text{rel}}\ ) \,\big| \\
&\qquad \mathit{Term}_{\text{rel}}+ \,\big|\, \mathit{Term}_{\text{rel}}\sim\ \mid\ \{\ \mathit{Term}_{\text{scalar}}\ \text{->}\ \mathit{Term}_{\text{set}}\ \}
\end{aligned}$$

As an overview, Un is the universe of atomic elements, & is the intersection operator, <: is the domain restriction operator, ; is the composition operator, + is the transitive closure operator, ~ is the inverse operator, Id is the identity relation, and $\mathit{Term}_{\text{rel}}(\mathit{Term}_{\text{set}})$ gives the relational image. The complete, formal definitions of the operators begin on page 16.

Atomic formulae are constructed from terms.

[1] For clarity of presentation, I did choose to retain some redundant operators. For example, in the presence of the inverse operator, only one of the dom and ran operators is necessary.

***Definition 2.9** *(AtomicFormula)*

$$\begin{aligned} \mathit{AtomicFormula} ::= {} & \mathit{Term}_{\text{scalar}} \text{ in } \mathit{Term}_{\text{set}} \mid \mathit{Term}_{\text{scalar}} = \mathit{Term}_{\text{scalar}} \mid \\ & \mathit{Term}_{\text{set}} = \mathit{Term}_{\text{set}} \mid \mathit{Term}_{\text{set}} <= \mathit{Term}_{\text{set}} \mid \\ & \mathit{Term}_{\text{rel}} = \mathit{Term}_{\text{rel}} \mid \mathit{Term}_{\text{rel}} <= \mathit{Term}_{\text{rel}} \mid \\ & \text{func } \mathit{Term}_{\text{rel}} \mid \text{true} \mid \text{false} \end{aligned}$$

The <= operator denotes the standard subset relationship.

Finally, Wffs in the formula language are built from atomic formulae.

***Definition 2.10** *(Wff)*

$$\mathit{Wff} ::= \mathit{AtomicFormula} \mid \text{not } \mathit{Wff} \mid (\ \mathit{Wff} \text{ and } \mathit{Wff}\ ) \mid (\ \mathit{Wff} \text{ or } \mathit{Wff}\ )$$

For any formula in the relational language, there is a unique derivation for that formula from this grammar. Formula (2.1) is the formula derived from the claim uniqueAddrAlloc given in Figure 1.1.[2]

(2.1)

```
( not ( (dom usage = used and dom usage' = used') and ( func usage and func usage' ) ) or
( not ( ( dom usage = used and dom usage' = used' ) and
     ( ( used <: usage' ) = usage and used' = ( used U {newAddr} ) ) ) or
not newAddr in used ) )
```

The set of variables for a term is given by the function *Var.*

Definition 2.11 **(Variables of a Term)**

The variables of a term are given by the function, $\mathit{Var}(\tau) : \mathit{Term} \longrightarrow \mathbf{P}\,\mathit{Variable}$, defined as all variables appearing in the term.

Similarly, the *Var* function yields the set of variables for a formula.

Definition 2.12 **(Variables of a Formula)**

The variable of a formula are given by the function, $\mathit{Var}(\phi) : \mathit{Wff} \longrightarrow \mathbf{P}\,\mathit{Variable}$, defined as all variables appearing in the formula.

For any given assignment in the context of some subset of *Value,* the interpretation of a formula is TRUE, FALSE, or UNK (unknown). Intuitively, the interpretation of a formula may be unknown if any of the variables of the formula are not mapped by the assignment. Otherwise, the interpretation function replaces each variable in the formula by the corresponding value from the assignment and evaluates the formula using the usual semantics for relational formulae.

The interpretation of a formula depends on the interpretation of each term. The notation $M_{\text{term}}[\tau,\alpha,\upsilon]$ describes the interpretation of a term τ in the formula language for an assignment α and a set of values υ. The set of values υ is a subset of *Value*. The interpretation of terms depends on the set of values because the language includes constructs such as Un and Id. The complete term interpretation function M_{term} is the union of three functions: M_{scalar}, M_{set}, and M_{rel}.

***Definition 2.13** **(M_{scalar})**

The interpretation of a scalar term τ for an assignment a and a set of values υ is given by the function $M_{\text{scalar}}[\tau,\alpha,\upsilon] : \mathit{Term}_{\text{scalar}} \times A \times \mathbf{P}\,\mathit{Value} \longrightarrow \mathit{Value}_{\text{scalar}} \cup \{\ \text{UNK}\ \}$,

[2] To simplify the presentation, I have elided the constraints that enforce the given type constraints. Removing these constraints from a well-typed formula does not change its satisfiability. As noted in Chapter 6, however, the implementation of Ladybug does make significant use of these constraints.

defined as

if τ is v where $v \in \mathit{Var}_{\text{scalar}}$

$$\begin{array}{ll} \alpha(v) & \text{if } v \in \operatorname{dom} \alpha \\ \text{UNK} & \text{otherwise} \end{array}$$

***Definition 2.14** **(M_{set})**

The interpretation of a set term τ for an assignment α and a set of values υ is given by the function $M_{\text{set}}[\tau,\alpha,\upsilon] : \mathit{Term}_{\text{set}} \times A \times \mathbb{P}\mathit{Value} \longrightarrow \mathit{Value}_{\text{set}} \cup \{ \text{UNK} \}$, defined as

if τ is v where $v \in \mathit{Var}_{\text{set}}$

$$\begin{array}{ll} \alpha(v) & \text{if } v \in \operatorname{dom} \alpha \\ \text{UNK} & \text{otherwise} \end{array}$$

if τ is $\{ \tau_1 \}$ where $\tau_1 \in \mathit{Term}_{\text{scalar}}$

$$\begin{array}{ll} \{ M_{\text{scalar}}[\tau_1,\alpha,\upsilon] \} & \text{if } M_{\text{scalar}}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \\ \text{UNK} & \text{otherwise} \end{array}$$

if τ is $\{ \}$

$$\varnothing$$

if τ is Un

$$U \cap \upsilon$$

if τ is τ_1 U τ_2 where $\tau_1,\tau_2 \in \mathit{Term}_{\text{set}}$

$$\begin{array}{ll} \{ x \mid x \in M_{\text{set}}[\tau_1,\alpha,\upsilon] \vee & \text{if } M_{\text{set}}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \wedge M_{\text{set}}[\tau_2,\alpha,\upsilon] \neq \text{UNK} \\ \qquad x \in M_{\text{set}}[\tau_2,\alpha,\upsilon] \} & \\ \text{UNK} & \text{otherwise} \end{array}$$

if τ is τ_1 & τ_2 where $\tau_1,\tau_2 \in \mathit{Term}_{\text{set}}$

$$\begin{array}{ll} \{ x \mid x \in M_{\text{set}}[\tau_1,\alpha,\upsilon] \wedge & \text{if } M_{\text{set}}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \wedge M_{\text{set}}[\tau_2,\alpha,\upsilon] \neq \text{UNK} \\ \qquad x \in M_{\text{set}}[\tau_2,\alpha,\upsilon] \} & \\ \text{UNK} & \text{otherwise} \end{array}$$

if τ is $\tau_1 \setminus \tau_2$ where $\tau_1,\tau_2 \in \mathit{Term}_{\text{set}}$

$$\begin{array}{ll} \{ x \mid x \in M_{\text{set}}[\tau_1,\alpha,\upsilon] \wedge & \text{if } M_{\text{set}}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \wedge M_{\text{set}}[\tau_2,\alpha,\upsilon] \neq \text{UNK} \\ \qquad x \notin M_{\text{set}}[\tau_2,\alpha,\upsilon] \} & \\ \text{UNK} & \text{otherwise} \end{array}$$

if τ is dom τ_1 where $\tau_1 \in \mathit{Term}_{\text{rel}}$

$$\begin{array}{ll} \{ x \mid \exists y.\ (x,y) \in M_{\text{rel}}[\tau_1,\alpha,\upsilon] \} & \text{if } M_{\text{rel}}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \\ \text{UNK} & \text{otherwise} \end{array}$$

if τ is ran τ_1 where $\tau_1 \in \mathit{Term}_{\text{rel}}$

$$\begin{array}{ll} \{ y \mid \exists x.\ (x,y) \in M_{\text{rel}}[\tau_1,\alpha,\upsilon] \} & \text{if } M_{\text{rel}}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \\ \text{UNK} & \text{otherwise} \end{array}$$

if τ is $\tau_1(\tau_2)$ where $\tau_1 \in \mathit{Term}_{\text{rel}}$ and $\tau_2 \in \mathit{Term}_{\text{set}}$

$$\begin{array}{ll} \{ y \mid \exists x.\ x \in M_{\text{set}}[\tau_2,\alpha,\upsilon] \wedge & \text{if } M_{\text{rel}}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \wedge \\ \qquad (x,y) \in M_{\text{rel}}[\tau_1,\alpha,\upsilon] \} & \qquad M_{\text{set}}[\tau_2,\alpha,\upsilon] \neq \text{UNK} \\ \text{UNK} & \text{otherwise} \end{array}$$

***Definition 2.15** **(M_{rel})**

The interpretation of a relational term τ for an assignment α and a set of values υ is given by the function $M_{\text{rel}}[\tau,\alpha,\upsilon] : \mathit{Term}_{\text{rel}} \times A \times \mathbb{P}\mathit{Value} \longrightarrow \mathit{Value}_{\text{rel}} \cup \{ \text{UNK} \}$, defined as

if τ is v where $v \in \mathit{Var}_{\text{rel}}$

$$\begin{array}{ll} \alpha(v) & \text{if } v \in \operatorname{dom} \alpha \\ \text{UNK} & \text{otherwise} \end{array}$$

if τ is τ_1 U τ_2 where $\tau_1,\tau_2 \in \mathit{Term}_{\text{rel}}$

$$\begin{array}{ll} \{ (x,y) \mid (x,y) \in M_{\text{rel}}[\tau_1,\alpha,\upsilon] \vee & \text{if } M_{\text{rel}}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \wedge M_{\text{rel}}[\tau_2,\alpha,\upsilon] \neq \text{UNK} \\ \qquad (x,y) \in M_{\text{rel}}[\tau_2,\alpha,\upsilon] \} & \\ \text{UNK} & \text{otherwise} \end{array}$$

if τ is Id

$$\{ (x,x) \mid x \in U \cap \upsilon \}$$

if τ is τ_1 & τ_2 where $\tau_1,\tau_2 \in Term_{\text{rel}}$

$$\begin{cases} \{ (x,y) \mid (x,y) \in M_{\text{rel}}[\tau_1,\alpha,\upsilon] \wedge \\ \quad (x,y) \in M_{\text{rel}}[\tau_2,\alpha,\upsilon] \} & \text{if } M_{\text{rel}}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \wedge M_{\text{rel}}[\tau_2,\alpha,\upsilon] \neq \text{UNK} \\ \text{UNK} & \text{otherwise} \end{cases}$$

if τ is $\tau_1 \setminus \tau_2$ where $\tau_1,\tau_2 \in Term_{\text{rel}}$

$$\begin{cases} \{ (x,y) \mid (x,y) \in M_{\text{rel}}[\tau_1,\alpha,\upsilon] \wedge \\ \quad (x,y) \notin M_{\text{rel}}[\tau_2,\alpha,\upsilon] \} & \text{if } M_{\text{rel}}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \wedge M_{\text{rel}}[\tau_2,\alpha,\upsilon] \neq \text{UNK} \\ \text{UNK} & \text{otherwise} \end{cases}$$

if τ is τ_1 ; τ_2 where $\tau_1,\tau_2 \in Term_{\text{rel}}$

$$\begin{cases} \{ (x,y) \mid \exists z.\ (x,z) \in M_{\text{rel}}[\tau_1,\alpha,\upsilon] \wedge \\ \quad (z,y) \in M_{\text{rel}}[\tau_2,\alpha,\upsilon] \} & \text{if } M_{\text{rel}}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \wedge M_{\text{rel}}[\tau_2,\alpha,\upsilon] \neq \text{UNK} \\ \text{UNK} & \text{otherwise} \end{cases}$$

if τ is τ_1+ where $\tau_1 \in Term_{\text{rel}}$

$$\begin{cases} \{ (x,y) \mid \exists z_1,z_2,\ldots,z_k. \\ \quad (x,z_1) \in M_{\text{rel}}[\tau_1,\alpha,\upsilon] \wedge \\ \quad (z_1,z_2) \in M_{\text{rel}}[\tau_1,\alpha,\upsilon] \wedge \ldots \\ \quad \wedge\ (z_k,y) \in M_{\text{rel}}[\tau_1,\alpha,\upsilon] \} & \text{if } M_{\text{rel}}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \\ \text{UNK} & \text{otherwise} \end{cases}$$

if τ is τ_1~ where $\tau_1 \in Term_{\text{rel}}$

$$\begin{cases} \{ (x,y) \mid (y,x) \in M_{\text{rel}}[\tau_1,\alpha,\upsilon] \} & \text{if } M_{\text{rel}}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \\ \text{UNK} & \text{otherwise} \end{cases}$$

if τ is τ_1 <: τ_2 where $\tau_1 \in Term_{\text{set}}$ and $\tau_2 \in Term_{\text{rel}}$

$$\begin{cases} \{ (x,y) \mid x \in M_{\text{set}}[\tau_1,\alpha,\upsilon] \wedge \\ \quad (x,y) \in M_{\text{rel}}[\tau_2,\alpha,\upsilon] \} & \text{if } M_{\text{set}}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \wedge \\ & \quad M_{\text{rel}}[\tau_2,\alpha,\upsilon] \neq \text{UNK} \\ \text{UNK} & \text{otherwise} \end{cases}$$

if τ is { τ_1 -> τ_2 } where $\tau_1 \in Term_{\text{scalar}}$ and $\tau_2 \in Term_{\text{set}}$

$$\begin{cases} \{ (x,y) \mid x = M_{\text{scalar}}[\tau_1,\alpha,\upsilon] \wedge \\ \quad y \in M_{\text{set}}[\tau_2,\alpha,\upsilon] \} & \text{if } M_{\text{scalar}}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \wedge \\ & \quad M_{\text{set}}[\tau_2,\alpha,\upsilon] \neq \text{UNK} \\ \text{UNK} & \text{otherwise} \end{cases}$$

***Definition 2.16** **(M_{term})**

The interpretation of a term τ for an assignment α and a set of values υ is given by the function $M_{\text{term}}[\tau,\alpha,\upsilon] : Term \times A \times \mathbf{P}\,Value \longrightarrow Value \cup \{ \text{UNK} \}$, defined by the union of the three specific interpretation functions:

$M_{\text{term}} = M_{\text{scalar}} \cup M_{\text{set}} \cup M_{\text{rel}}$

The interpretation of a formula for an assignment in the context of a set of values is given by $M[\phi,\alpha,\upsilon]$, where $\phi \in Wff$, $\alpha \in A$, and $\upsilon \subseteq Value$.

***Definition 2.17** **(M)**

The interpretation of a formula ϕ for an assignment α is given by the function $M[\phi,\alpha,\upsilon] : Wff \times A \times \mathbf{P}\,Value \longrightarrow \{ \text{TRUE}, \text{FALSE}, \text{UNK} \}$, defined as

if ϕ is $\tau_1 = \tau_2$ where $\tau_1,\tau_2 \in Term_{\text{scalar}}$

$$\begin{cases} \text{TRUE} & \text{if } M_{\text{scalar}}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \wedge M_{\text{scalar}}[\tau_2,\alpha,\upsilon] \neq \text{UNK} \wedge \\ & \quad M_{\text{scalar}}[\tau_1,\alpha,\upsilon] = M_{\text{scalar}}[\tau_2,\alpha,\upsilon]) \\ \text{FALSE} & \text{if } M_{\text{scalar}}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \wedge M_{\text{scalar}}[\tau_2,\alpha,\upsilon] \neq \text{UNK} \wedge \\ & \quad M_{\text{scalar}}[\tau_1,\alpha,\upsilon] \neq M_{\text{scalar}}[\tau_2,\alpha,\upsilon] \\ \text{UNK} & \text{otherwise} \end{cases}$$

if ϕ is τ_1 in τ_2 where $\tau_1 \in \mathit{Term}_{\mathrm{scalar}}$ and $\tau_2 \in \mathit{Term}_{\mathrm{set}}$

TRUE if $M_{\mathrm{scalar}}[\tau_1,\alpha,\upsilon] \neq \mathrm{UNK} \wedge M_{\mathrm{set}}[\tau_2,\alpha,\upsilon] \neq \mathrm{UNK} \wedge$
$M_{\mathrm{scalar}}[\tau_1,a,\upsilon] \in M_{\mathrm{set}}[\tau_2,\alpha,\upsilon]$

FALSE if $M_{\mathrm{scalar}}[\tau_1,\alpha,\upsilon] \neq \mathrm{UNK} \wedge M_{\mathrm{set}}[\tau_2,\alpha,\upsilon] \neq \mathrm{UNK} \wedge$
$M_{\mathrm{scalar}}[\tau_1,\alpha,\upsilon] \notin M_{\mathrm{set}}[\tau_2,\alpha,\upsilon]$

UNK otherwise

if ϕ is $\tau_1 = \tau_2$ where $\tau_1,\tau_2 \in \mathit{Term}_{\mathrm{set}}$

TRUE if $M_{\mathrm{set}}[\tau_1,\alpha,\upsilon] \neq \mathrm{UNK} \wedge M_{\mathrm{set}}[\tau_2,\alpha,\upsilon] \neq \mathrm{UNK} \wedge$
$M_{\mathrm{set}}[\tau_1,\alpha,\upsilon] = M_{\mathrm{set}}[\tau_2,\alpha,\upsilon]$

FALSE if $M_{\mathrm{set}}[\tau_1,\alpha,\upsilon] \neq \mathrm{UNK} \wedge M_{\mathrm{set}}[\tau_2,\alpha,\upsilon] \neq \mathrm{UNK} \wedge$
$M_{\mathrm{set}}[\tau_1,\alpha,\upsilon] \neq M_{\mathrm{set}}[\tau_2,\alpha,\upsilon]$

UNK otherwise

if ϕ is τ_1 <= τ_2 where $\tau_1,\tau_2 \in \mathit{Term}_{\mathrm{set}}$

TRUE if $M_{\mathrm{set}}[\tau_1,\alpha,\upsilon] \neq \mathrm{UNK} \wedge M_{\mathrm{set}}[\tau_2,\alpha,\upsilon] \neq \mathrm{UNK} \wedge$
$M_{\mathrm{set}}[\tau_1,\alpha,\upsilon] \subseteq M_{\mathrm{set}}[\tau_2,\alpha,\upsilon]$

FALSE if $M_{\mathrm{set}}[\tau_1,\alpha,\upsilon] \neq \mathrm{UNK} \wedge M_{\mathrm{set}}[\tau_2,\alpha,\upsilon] \neq \mathrm{UNK} \wedge$
$\exists x \in M_{\mathrm{set}}[\tau_1,\alpha,\upsilon].\ x \notin M_{\mathrm{set}}[\tau_2,\alpha,\upsilon]$

UNK otherwise

if ϕ is $\tau_1 = \tau_2$ where $\tau_1,\tau_2 \in \mathit{Term}_{\mathrm{rel}}$

TRUE if $M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \neq \mathrm{UNK} \wedge M_{\mathrm{rel}}[\tau_2,\alpha,\upsilon] \neq \mathrm{UNK} \wedge$
$M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] = M_{\mathrm{rel}}[\tau_2,\alpha,\upsilon]$

FALSE if $M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \neq \mathrm{UNK} \wedge M_{\mathrm{rel}}[\tau_2,\alpha,\upsilon] \neq \mathrm{UNK} \wedge$
$M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \neq M_{\mathrm{rel}}[\tau_2,\alpha,\upsilon]$

UNK otherwise

if ϕ is τ_1 <= τ_2 where $\tau_1,\tau_2 \in \mathit{Term}_{\mathrm{rel}}$

TRUE if $M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \neq \mathrm{UNK} \wedge M_{\mathrm{rel}}[\tau_2,\alpha,\upsilon] \neq \mathrm{UNK} \wedge$
$M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \subseteq M_{\mathrm{rel}}[\tau_2,\alpha,\upsilon]$

FALSE if $M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \neq \mathrm{UNK} \wedge M_{\mathrm{rel}}[\tau_2,\alpha,\upsilon] \neq \mathrm{UNK} \wedge$
$\exists (x,y) \in M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon].\ (x,y) \notin M_{\mathrm{rel}}[\tau_2,\alpha,\upsilon]$

UNK otherwise

if ϕ is func τ_1 where $\tau_1 \in \mathit{Term}_{\mathrm{rel}}$

TRUE if $M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \neq \mathrm{UNK} \wedge$
$\forall x,y,z.((x,y) \in M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \wedge (x,z) \in M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon]) \Rightarrow y = z$

FALSE if $M_{\mathrm{rel}}[\tau_1,a,\upsilon] \neq \mathrm{UNK} \wedge$
$\exists x,y,z.(x,y) \in M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \wedge (x,z) \in M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \wedge y \neq z$

UNK otherwise

if ϕ is true

TRUE

if ϕ is false

FALSE

if ϕ is (ϕ_1 and ϕ_2) where $\phi_1,\phi_2 \in \mathit{Wff}$

TRUE if $M[\phi_1,\alpha,\upsilon] = \mathrm{TRUE} \wedge M[\phi_2,\alpha,\upsilon] = \mathrm{TRUE}$

FALSE if $M[\phi_1,\alpha,\upsilon] = \mathrm{FALSE} \vee M[\phi_2,\alpha,\upsilon] = \mathrm{FALSE}$

UNK otherwise

if ϕ is (ϕ_1 or ϕ_2) where $\phi_1,\phi_2 \in \mathit{Wff}$

TRUE if $M[\phi_1,\alpha,\upsilon] = \mathrm{TRUE} \vee M[\phi_2,\alpha,\upsilon] = \mathrm{TRUE}$

FALSE if $M[\phi_1,\alpha,\upsilon] = \mathrm{FALSE} \wedge M[\phi_2,\alpha,\upsilon] = \mathrm{FALSE}$

UNK otherwise

if ϕ is not ϕ_1 where $\phi_1 \in \mathit{Wff}$
    TRUE   if $M[\phi_1,\alpha,\upsilon] =$ FALSE
    FALSE   if $M[\phi_1,\alpha,\upsilon] =$ TRUE
    UNK   otherwise

Definitions of the term and formula interpretation functions must be a part of the language definition for any problem domain.

As an example of the relational formula interpretation function, consider the negation of (2.1) (derived from uniqueAddrAlloc) given by (2.2):

(2.2)

```
( ( ( dom usage = used and dom usage' = used' ) and
      ( func usage and func usage' ) ) and
      ( ( ( used <: usage' ) = usage and used' = ( used U {newAddr} ) ) and
      newAddr in used ) )
```

This formula can be interpreted with respect to any assignment and set of values. Using the subset of U called u, including elements a_0 and d_0, three possible interpretations are

$M[(2.2),\{\ \},u \cup \mathrm{P}u \cup \mathrm{P}(u\times u)] =$ UNK

$M[(2.2),\{ \text{usage}' \mapsto \{ a_0 \mapsto d_0 \}, \text{used}' \mapsto \varnothing \},u \cup \mathrm{P}u \cup \mathrm{P}(u\times u)] =$ FALSE

$M[(2.2),$
$\{ \text{usage}' \mapsto \{ a_0 \mapsto d_0 \}, \text{used}' \mapsto \{a_0\},\text{usage} \mapsto \{ a_0 \mapsto d_0 \}, \text{used} \mapsto \{a_0\}, \text{newAddr} \mapsto a_0 \},$
$u \cup \mathrm{P}u \cup \mathrm{P}(u\times u)] =$ TRUE

If the assignment maps all the variables in a term, the interpretation of the term for that assignment is an element of *Value*, not UNK.

Lemma 2.1 $\forall\upsilon \subseteq \mathit{Value}.\forall\alpha \in A.\ \forall\tau \in \mathit{Term}.\ \mathit{Var}(\tau) \subseteq \mathrm{dom}\ \alpha \Rightarrow M_{\text{term}}[\tau,\alpha,\upsilon] \neq$ UNK

Proof: By structural induction.

If τ is v where v $\in$ *Variable*
    Obviously, $\mathit{Var}(\tau) = \{ \text{v} \}$
    Because $\mathit{Var}(\tau) \subseteq \mathrm{dom}\ \alpha$, v $\in \mathrm{dom}\ \alpha$
    By definition of M_{term}, v $\in \mathrm{dom}\ \alpha \Rightarrow M_{\text{term}}[\tau,\alpha,\upsilon] \neq$ UNK.

if τ is τ_1 U τ_2 where $\tau_1,\tau_2 \in \mathit{Term}_{\text{set}}$
    Obviously, $\mathit{Var}(\tau_1) \subseteq \mathit{Var}(\tau)$ and $\mathit{Var}(\tau_2) \subseteq \mathit{Var}(\tau)$
    Therefore, $\mathit{Var}(\tau_1) \subseteq \mathrm{dom}\ \alpha$ and $\mathit{Var}(\tau_2) \subseteq \mathrm{dom}\ \alpha$
    Therefore, by hypothesis, $M_{\text{term}}[\tau_1,\alpha,\upsilon] \neq$ UNK and $M_{\text{term}}[\tau_2,\alpha,\upsilon] \neq$ UNK
    By definition of M_{term}, $M_{\text{term}}[\tau,\alpha,\upsilon] \neq$ UNK

Other productions follow similarly. ■

Similarly, if the assignment maps all the variables in a formula, the interpretation of the formula for the assignment is either TRUE or FALSE, but not UNK.

Lemma 2.2 $\forall\alpha \in A.\ \forall\phi \in \mathit{Wff}.\ \mathit{Var}(\phi) \subseteq \mathrm{dom}\ \alpha \Rightarrow M[\phi,\alpha,\upsilon] \neq$ UNK

Proof: By structural induction.

If ϕ is τ_1 in τ_2 where $\tau_1 \in \mathit{Term}_{\text{scalar}}$ and $\tau_2 \in \mathit{Term}_{\text{set}}$
    Obviously, $\mathit{Var}(\tau_1) \subseteq \mathit{Var}(\phi)$ and $\mathit{Var}(\tau_2) \subseteq \mathit{Var}(\phi)$
    Therefore, $\mathit{Var}(\tau_1) \subseteq \mathrm{dom}\ \alpha$ and $\mathit{Var}(\tau_2) \subseteq \mathrm{dom}\ \alpha$
    Therefore, by Lemma 2.1, $M_{\text{term}}[\tau_1,\alpha,\upsilon] \neq$ UNK and $M_{\text{term}}[\tau_2,\alpha,\upsilon] \neq$ UNK

By definition of M, $M[\phi,\alpha,\upsilon] \neq$ UNK

If ϕ is (ϕ_1 and ϕ_2) where $\phi_1,\phi_2 \in Wff$
Obviously, $Var(\phi_1) \subseteq Var(\phi)$ and $Var(\phi_2) \subseteq Var(\phi)$
Therefore, $Var(\phi_1) \subseteq$ dom α and $Var(\phi_2) \subseteq$ dom α
Therefore, by hypothesis, $M[\phi_1,\alpha,\upsilon] \neq$ UNK and $M[\phi_2,\alpha,\upsilon] \neq$ UNK
By definition of M, $M[\phi,\alpha,\upsilon] \neq$ UNK

Other productions follow similarly. ■

Lemma 2.1 and Lemma 2.2 must hold for the language definition for any valid problem domain.

Some formulae are logically entailed by other formulae. The notation $\phi \models \phi'$ indicates that ϕ logically entails ϕ'.

Definition 2.18 **(Logical Entailment)**
$\phi \models \phi'$ iff $\forall \upsilon \subseteq Value. \forall \alpha \in A.\ M[\phi,\alpha,\upsilon] = \text{TRUE} \Rightarrow M[\phi',\alpha,\upsilon] = \text{TRUE}$

For the formula (2.2) given earlier, newAddr must be an element of used for any satisfying assignment.

(2.2)⊨newAddr in used

2.3 Search

Selective enumeration is a generate-and-test search that finds a solution to a formula. This section formally defines a search and its solution.

Selective enumeration orders the variables to control the order of the search. Some search approaches determine the order that variables will be considered dynamically, choosing the next variable based on the partial assignment generated thus far. Selective enumeration, on the other hand, fixes the order for the entire search prior to starting the search. The function *Ord* defines this ordering.

Definition 2.19 **(*Ord*)**
The variables are ordered according to *Ord*, a one-to-one function mapping the variables to the first *N* natural numbers.

The search illustrated by Figure 1.2 and Figure 1.3 uses the ordering

Ord = { usage' ↦ 1, used' ↦ 2, usage ↦ 3, used ↦ 4, newAddr ↦ 5 }

The search considers only subsets of the variables at a time, choosing these subsets based on the ordering. The i^{th} subset of *Variable* contains the first *i* variables, as determined by *Ord*.

Definition 2.20 **(Var_i)**
$Var_i = \{ v \in Variable \mid 1 \leqslant Ord(v) \leqslant i \}$

Projecting these subsets of *Variable* though *A* yields sets of assignments that prove useful in describing selective enumeration.

Definition 2.21 **(A_i)**
$A_i = \{ \alpha \in A \mid \text{dom } \alpha = Var_i \}$

A_i is the set of assignments that map exactly the first *i* variables, as defined by *Ord*. The assignments on the i^{th} level of the search tree are drawn from A_i. Two such sets are of particular note: A_0 contains only the empty assignment and A_N contains all full assignments.

A *solution* is a satisfying full assignment.

Definition 2.22 **(Solution)**
A full assignment $\alpha \in A_N$ is a *solution* for a formula ϕ and a set of values υ iff

$$M[\phi,\alpha,\upsilon] = \text{TRUE}.$$

A *search* is a procedure that discovers solutions to a formula. The search is limited to the subset of *Value* given by υ.

Definition 2.23 **(Search)**
A function $\omega(\phi,\upsilon) : \mathit{Wff} \times \mathbb{P}\,\mathit{Value} \longrightarrow A_N$ is a *search* iff

$$\forall \alpha \in \omega(\phi,\upsilon).\ \text{ran}\ \alpha \subseteq \upsilon \wedge M[\phi,\alpha,\upsilon] = \text{TRUE}.$$

A search is *sound* if it is guaranteed to find a solution if any exist for the formula and the set of values considered.

Definition 2.24 **(Sound Search)**
A search $\omega(\phi,\upsilon)$ is *sound* for a formula ϕ and a set of values υ iff

$$(\exists a \in A_N.\ M[\phi,\alpha,\upsilon] = \text{TRUE} \wedge \text{ran}\ \alpha \subseteq \upsilon) \Rightarrow \omega(\phi,\upsilon) \neq \emptyset.$$

A search is *complete,* on the other hand, if it discovers all solutions to a formula. Completeness can be an undesirable property, as the number of solutions to a formula may overwhelm their intended consumer.

2.4 Duplications

The essence of selective enumeration is reducing the number of cases to be tested by not generating "duplicates". A *duplication* is an equivalence relation on the full assignments, partitioning them into equivalence classes for a formula ϕ. All assignments in any equivalence class must give the same interpretation to ϕ. Selective enumeration generates at least one assignment to be tested from each equivalence class. Assignments beyond the first in any equivalence class are the duplicates to be suppressed.

Definition 2.25 **(Duplication)**
An equivalence relation $\approx_d$ is a *duplication* for the formula ϕ and set of values υ iff

$$\forall \alpha,\alpha' \in A_N.\ \text{ran}\ \alpha \subseteq \upsilon \wedge \text{ran}\ \alpha' \subseteq \upsilon \wedge \alpha \approx_d \alpha' \Rightarrow M[\phi,\alpha,\upsilon] = M[\phi,\alpha',\upsilon]$$

Two obvious duplications are uninteresting for the purposes of selective enumeration. The first, which I call $\approx_\emptyset$, places each assignment in its own equivalence class. This corresponds to the exhaustive-enumeration search. The other obvious duplication, which I call $\approx_M$, divides the assignments into two classes, ones that satisfy ϕ and ones that do not satisfy ϕ. Although this would be the ideal duplication, it is not directly computable for interesting problem domains and therefore of little practical benefit. These two duplications are formally defined as

(2.3) $$\forall \alpha,\alpha' \in A_N.\ \alpha \approx_\emptyset \alpha' \Leftrightarrow \alpha = \alpha'$$

(2.4) $$\forall \upsilon \subseteq \mathit{Value}. \forall \alpha,\alpha' \in A_N.\ \text{ran}\ \alpha \subseteq \upsilon \wedge \text{ran}\ \alpha' \subseteq \upsilon \Rightarrow (\alpha \approx_M \alpha' \Leftrightarrow M[\phi,\alpha,\upsilon] = M[\phi,\alpha',\upsilon])$$

The duplications described in the previous chapter for finding the counterexample to uniqueAddrAlloc can also be defined in this manner. The clearest form of duplication exploited by Ladybug is the isomorph duplication: two full assignments are duplicates if they are isomorphic to each other. Chapter 4 formally defines this duplication and describes how it can be exploited by the search. Intuitively, any two isomorphic assignments will yield the same interpretation for the formula. The search will be sound if it ignores all but one assignment in each equivalence class defined by the duplication.

The other class of duplications exploited by Ladybug involve equating assignments that fail to satisfy a simple formula, called a filter formula, that is entailed by the formula being solved. These duplications are different than the isomorph duplication in a fundamental way: the interpretation of some equivalence classes is known a-priori. The search can eliminate all assignments in an equivalence class known to be not satisfying.

One such duplication highlighted in the previous example search exploited the filter formula newAddr in used, which is a conjunct of the formula being solved. For this reduction, all assignments for which newAddr is not an element of used form a single equivalence class, with each remaining assignment forming its own equivalence class.

(2.5)
$$\begin{aligned}&\forall \upsilon \subseteq \mathit{Value}.\forall \alpha,\alpha' \in A_N.\ \mathrm{ran}\ \alpha \subseteq \upsilon \wedge \mathrm{ran}\ \alpha' \subseteq \upsilon \Rightarrow\\ &\quad(\alpha \approx_{\mathrm{newAddr\ in\ used}} \alpha' \Leftrightarrow\\ &\qquad((\mathcal{M}[\mathrm{newAddr\ in\ used},\alpha,\upsilon] = \mathrm{FALSE} \wedge \mathcal{M}[\mathrm{newAddr\ in\ used},\alpha',\upsilon] = \mathrm{FALSE}) \vee \alpha = \alpha'))\end{aligned}$$

Other bounded generation duplications, such as the constraint on usage and usage', behave similarly. They group known false assignments together into a single equivalence class, placing all other assignments into individual equivalence classes.

This form of equivalence relation can be generalized to support any partial assignment duplication. Each partial assignment duplication has a filter formula ϕ' that is entailed by the target formula itself.

Definition 2.26 **(Partial Assignment Duplication)**

An equivalence relation $\approx_{\rho(\phi')}$ is a *partial assignment duplication* of the related formula ϕ' for a formula ϕ and a set of values υ iff

$$\phi \vDash \phi' \wedge\ \forall \alpha,\alpha' \in A_N.\ (\alpha \approx_{\rho(\phi')} \alpha' \Leftrightarrow (\alpha = \alpha' \vee (\mathcal{M}[\phi',\alpha,\upsilon] = \mathrm{FALSE} \wedge \mathcal{M}[\phi',\alpha',\upsilon] = \mathrm{FALSE})))$$

This equivalence relation places all assignments that fail to satisfy ϕ' into a common equivalence class and each assignment that satisfies ϕ' into its own equivalence class.

Lemma 2.3 The partial assignment duplication $\approx_{\rho(\phi')}$ is a duplication for the formula ϕ and the set of values υ.

Proof: To prove that $\approx_{\rho(\phi')}$ is a duplication, it is necessary to prove that

$$\forall \alpha,\alpha' \in A_N.\mathrm{ran}\ \alpha \subseteq \upsilon \wedge \mathrm{ran}\ \alpha' \subseteq \upsilon \wedge \alpha \approx_{\rho(\phi')} \alpha' \Rightarrow \mathcal{M}[\phi,\alpha,\upsilon] = \mathcal{M}[\phi,\alpha',\upsilon]$$

By the definition of $\approx_{\rho(\phi')}$, there are two cases to consider:

(1) $\alpha = \alpha'$

(2) $(\mathcal{M}[\phi',\alpha,\upsilon] = \mathrm{FALSE} \wedge \mathcal{M}[\phi',\alpha',\upsilon] = \mathrm{FALSE})$

For (1), clearly $\mathcal{M}[\phi,\alpha,\upsilon] = \mathcal{M}[\phi,\alpha',\upsilon]$.

For (2), because $\alpha \in A_N$, $\mathcal{M}[\phi,\alpha,\upsilon]$ is either TRUE or FALSE by Lemma 2.2.
Because $\phi \vDash \phi'$, if $\mathcal{M}[\phi',\alpha,\upsilon] = \mathrm{FALSE}$, then $\mathcal{M}[\phi,\alpha,\upsilon] = \mathrm{FALSE}$.
Therefore, for (2), $\mathcal{M}[\phi,\alpha,\upsilon] = \mathrm{FALSE} \wedge \mathcal{M}[\phi,\alpha',\upsilon] = \mathrm{FALSE}$. ■

Duplications may also be combined, resulting in a duplication with generally fewer equivalence classes (but never more). The combination of two duplications is the transitive closure of the union of the two duplications.

Definition 2.27 **(Duplication Composition)**

For any duplications $\approx_a$ and $\approx_b$ for a formula ϕ and a set of values υ, their composition, $\approx_a \circ \approx_b$, is defined as the smallest relation such that

$\forall\alpha,\alpha' \in A_N.\ (\alpha \approx_a \circ \approx_b \alpha' \Leftrightarrow$

$(\alpha \approx_a \alpha' \vee \alpha \approx_b \alpha' \vee (\exists\alpha'' \in A_N.(\alpha \approx_a \circ \approx_b \alpha'' \wedge \alpha'' \approx_a \circ \approx_b \alpha'))))$

The result of combining two duplications is itself an equivalence relation and therefore a duplication.

Lemma 2.4 For any duplications $\approx_a$ and $\approx_b$ for a formula ϕ and a set of values υ, $\approx_a \circ \approx_b$ is an equivalence relation.

Proof: A relation must be reflexive, symmetric, and transitive to be an equivalence relation.

Let $\approx_{ab} = \approx_a \circ \approx_b$

There are three possibilities allowed by the definition of $\approx_a \circ \approx_b$

(1) $\alpha \approx_a \alpha'$
(2) $\alpha \approx_b \alpha'$
(3) $\exists\alpha'' \in A_N.\ \alpha \approx_{ab} \alpha'' \wedge \alpha'' \approx_{ab} \alpha' \Rightarrow \alpha \approx_{ab} \alpha'$

Therefore, $\alpha \approx_a \alpha' \Rightarrow \alpha \approx_{ab} \alpha'$ and $\alpha \approx_b \alpha' \Rightarrow \alpha \approx_{ab} \alpha'$

As $\approx_a$ is reflexive, $\forall\alpha \in A_N.\ \alpha \approx_a \alpha$.
Therefore, because $\alpha \approx_a \alpha' \Rightarrow \alpha \approx_{ab} \alpha'$,
$\forall\alpha \in A_N.\ \alpha \approx_{ab} \alpha$ and $\approx_{ab}$ is reflexive

To demonstrate symmetry, I use structural induction with the hypothesis
$\forall\alpha,\alpha' \in A_N\ .\ \alpha \approx_{ab} \alpha' \Rightarrow \alpha' \approx_{ab} \alpha$
If (1) holds,
then $\alpha' \approx_a \alpha$ because $\approx_a$ is symmetrical.
Therefore, because $\alpha \approx_a \alpha' \Rightarrow \alpha \approx_{ab} \alpha'$,
$\alpha' \approx_{ab} \alpha$
A similar argument holds for (2)
If (3) holds, $\alpha \approx_{ab} \alpha''$ and $\alpha'' \approx_{ab} \alpha'$ for some $\alpha'' \in A_N$
By induction, $\alpha'' \approx_{ab} \alpha$ and $\alpha' \approx_{ab} \alpha''$
Therefore, $\alpha' \approx_{ab} \alpha'' \wedge \alpha'' \approx_{ab} \alpha$
Therefore, $\alpha' \approx_{ab} \alpha$
Therefore $\alpha' \approx_{ab} \alpha$ and $\approx_{ab}$ is symmetric.

Transitivity is a direct result of (3). ■

Lemma 2.5 For any duplications $\approx_a$ and $\approx_b$ for a formula ϕ and a set of values υ, $\approx_a \circ \approx_b$ is a duplication for ϕ and υ.

Proof: Let $\approx_{ab} = \approx_a \circ \approx_b$.

There are two requirements for $\approx_{ab}$ to be a duplication:

(a) $\approx_{ab}$ must be an equivalence relation on A_N.
(b) $\forall\upsilon \subseteq \mathit{Value}.\forall\phi \in \mathit{Wff}.\forall\alpha,\alpha' \in A_N.$
$\operatorname{ran} \alpha \subseteq \upsilon \wedge \operatorname{ran} \alpha' \subseteq \upsilon \wedge \alpha \approx_{ab} \alpha' \Rightarrow M[\phi,\alpha,\upsilon] = M[\phi,\alpha',\upsilon]$.

By Lemma 2.4, $\approx_{ab}$ is an equivalence relation on A_N.

Assume $\alpha,\alpha' \in A_N$ such that $\alpha \approx_{ab} \alpha'$

By definition, one of

(1) $\alpha \approx_a \alpha'$

(2) $\alpha \approx_b \alpha'$

(3) $\exists \alpha'' \in A_N.\ (\alpha \approx_{ab} \alpha'' \wedge \alpha'' \approx_{ab} \alpha')$

must hold.

If either (1) or (2) holds,
then $M[\phi,\alpha,\upsilon] = M[\phi,\alpha',\upsilon]$ because $\approx_a$ and $\approx_b$ are duplications.

(3) requires the existence of a sequence of full assignments,

$\alpha_1, \alpha_2, \ldots, \alpha_k \in A_N$.

$$\alpha \approx_1 \alpha_1 \wedge \alpha_1 \approx_2 \alpha_2 \wedge \ldots \wedge \alpha_{k-1} \approx_k \alpha_k \wedge \alpha_k \approx_{k+1} \alpha'$$

where $\approx_1, \approx_2, \ldots, \approx_k, \approx_{k+1}$ are either $\approx_a$ or $\approx_b$

That any such sequence must guarantee that $M[\phi,\alpha,\upsilon] = M[\phi,\alpha',\upsilon]$ is obvious by induction on the length of the sequence. ■

2.5 Generators

The key to any generate-and-test search is the ability to generate assignments. A special function, called a *generator*, generates a finite set of assignments for level i of the search tree given an *initial assignment*, which I refer to as α_0, from level *i-1* and a finite set of values, which I refer to as υ. For each assignment generated, a level i generator binds a value from υ to the i^{th} variable, copying the mapping of the first *i-1* variables from the initial assignment α_0.

Definition 2.28 **(Generator)**

A function $g : A_{i-1} \times \mathbb{P}\mathit{Value} \rightarrow \mathbb{P}A_i$ is a level i *generator* iff

$$\forall \upsilon \subseteq \mathit{Value}. \forall \alpha_0 \in A_{i-1}.\ \forall \alpha \in g(\alpha_0,\upsilon).\ \mathit{Var}_{i-1} \lhd \alpha = \alpha_0 \wedge \alpha(v_i) \in (\upsilon \cap \mathit{Typing}(\ v_i)\ )$$

where $v_i \in \mathit{Variable}$ and $\mathit{Ord}(v_i) = i$.

A generator function expands a single assignment in the search tree into the set of its immediate children in the search tree. Therefore, the fanout of the search tree at each level (and the ultimate number of assignments generated) is dependent on the number of assignments generated by the generator for that level.

An assignment is possibly satisfying if any full assignment that contains it is satisfying. A generator is *sound* if the only possibly satisfying assignments excluded are duplicates of other assignments generated. In other words, when given the prefix of a satisfying assignment as its initial assignment, a sound generator will generate the prefix of a satisfying assignment that is equivalent under the duplication.

Definition 2.29 **(Sound Generator)**

A level i generator g is *sound* for a duplication $\approx_d$ iff

$$\forall \upsilon \subseteq \mathit{Value}. \forall \alpha \in A_N.\ (M[\phi,\alpha,\upsilon] = \text{TRUE} \wedge \operatorname{ran} \alpha \subseteq \upsilon) \Rightarrow$$
$$\exists \alpha' \in A_N.\ \mathit{Var}_i \lhd \alpha' \in g(\mathit{Var}_{i-1} \lhd \alpha,\upsilon) \wedge \alpha \approx_d \alpha' \wedge \operatorname{ran} \alpha' \subseteq \upsilon.$$

As a reminder, a duplication is an equivalence relation over full assignments such that two assignments can be in the same equivalence class only if they give the same interpretation to the formula. This definition of soundness for generators projects the duplication back to level i of the search tree using domain restriction. A generator is sound if, for each invocation, it excludes an assignment α only if (1) every full assignment extension of α gives a false interpretation to the formula or (2), for each equivalence class in which a satisfying full assignment extension of α occurs, the generator yields an assignment α' that has a full assignment extension in that equivalence

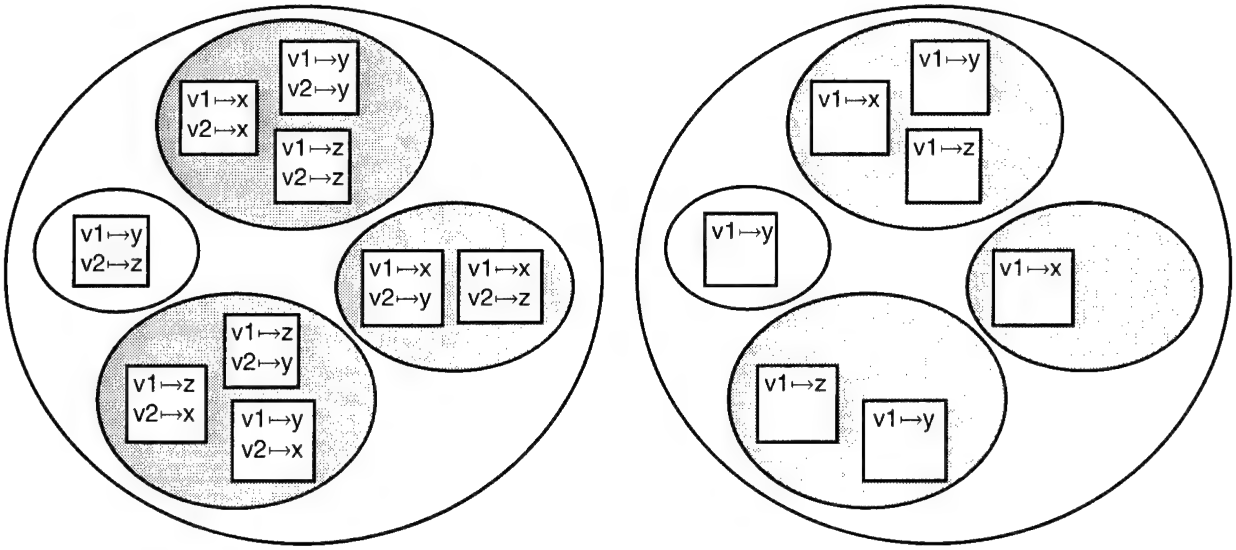


Figure 2.1. The left hand side illustrates a duplication for a simple two-variable search and the right hand side illustrates the projection of that duplication to level 1. The shaded equivalence classes contain satisfying assignments.

class.

Figure 2.1 illustrates a duplication for a simple two-variable search and the projection of that duplication into level 1. A sound level 1 generator can safely omit the partial assignment {v1↦y} if it generates the partial assignment {v1↦z}, as all satisfying equivalence classes that contain extensions of {v1↦y} also contain an extension of {v1↦z}. Therefore, only the partial assignments ({v1↦x} and {v1↦z}) must be generated for the generator to be sound.

For any level of any search of any formula for any set of values, there is at least one sound generator. A trivial level *i* generator, which I call *gx*, implements the naive exhaustive-enumeration search.

Definition 2.30 (*gx*)
The level *i* exhaustive-enumeration generator $gx : A_{i-1} \times \mathbf{P}\,Value \longrightarrow \mathbf{P}A_i$ is defined as

$$gx(\alpha_0,\upsilon) = \{\ \alpha_0 \cup \{ v_i \mapsto x \} \mid x \in (\upsilon \cap Typing(v_i))\ \}$$

where $v_i \in Variable.\ Ord(v_i) = i$.

Because the exhaustive-enumeration generator generates every possible assignment, it is sound, with many unnecessary assignments typically being generated.

Lemma 2.6 The exhaustive-enumeration generator *gx*, as defined in Definition 2.29, is sound for any duplication.

Proof: Obvious.

A search becomes unsound only if a satisfying equivalence class is *orphaned*, having no appropriately domain-projected assignments generated at some level of the search tree. For the duplication illustrated in Figure 2.1, failing to generate the partial assignment {v1↦x} at level 1 would orphan the rightmost equivalence class (the one containing {v1↦x,v2↦y} and {v1↦x,v2↦z}).

Assuming that the prior level of the search tree has not orphaned any satisfying equivalence classes, a search level expanded by a sound generator will not orphan any satisfying equivalence classes either. More precisely, at level *i*, and for any satisfying full assignment α, there must be an equivalent full assignment α' whose projection ($Var_{i-1} \lhd \alpha'$) was generated at level *i-1* if the equivalence class containing α was not already orphaned. A sound level *i* generator is guaranteed to generate the projection of some full assignment α'' that is equivalent to α, therefore guaranteeing that the equivalence class is still not orphaned.

Lemma 2.7 If a level *i* generator g is sound for $\approx_d$ then
$\forall \phi \in Wff.\ \forall \upsilon \subseteq Value.\ \forall \alpha, \alpha' \in A_N.\ (\alpha \approx_d \alpha' \wedge M[\phi,\alpha,\upsilon] = \text{TRUE} \wedge \text{ran } \alpha' \subseteq \upsilon) \Rightarrow$
$\exists \alpha'' \in A_N.\ \alpha \approx_d \alpha'' \wedge Var_i \lhd \alpha'' \in g(Var_{i-1} \lhd \alpha', \upsilon)$

Proof: Because g is sound,
$\exists \alpha'' \in A_N.\ \alpha' \approx_d \alpha'' \wedge Var_i \lhd \alpha'' \in g(Var_{i-1} \lhd \alpha', \upsilon)$.
By transitivity, $\alpha \approx_d \alpha''$. ■

A combination of duplications defines a new duplication, with each new equivalence class completely containing at least one equivalence class from each initial duplication. Therefore, a generator that is sound for one duplication is also sound for that duplication in combination with any other duplication.

Theorem 2.8 If a level *i* generator g is sound for some duplication $\approx_a$,
then for any duplication $\approx_b$, g is also sound for $\approx_a \circ \approx_b$.

Proof: Let $\upsilon \subseteq Value$ and $\alpha \in A_N$ such that $\text{ran } \alpha \subseteq \upsilon \wedge M[\phi,\alpha,\upsilon] = \text{TRUE}$.
Also let $\approx_{ab} = \approx_a \circ \approx_b$.
By the definition of a sound generator,
$\exists \alpha' \in A_N. Var_i \lhd \alpha' \in g(Var_{i-1} \lhd \alpha, \upsilon) \wedge \alpha \approx_a \alpha' \wedge \text{ran } \alpha' \subseteq \upsilon$.
By definition of $\circ$, $\alpha \approx_a \alpha' \Rightarrow \alpha \approx_{ab} \alpha'$.
Therefore, $\exists \alpha' \in A_N. Var_i \lhd \alpha' \in g(Var_{i-1} \lhd \alpha, \upsilon) \wedge \alpha \approx_{ab} \alpha' \wedge \text{ran } \alpha' \subseteq \upsilon$.
Therefore, g is sound for $\approx_a \circ \approx_b$. ■

A search requires a collection of generators, one associated with each variable. A function mapping each variable to an appropriate generator is called a generator suite.

Definition 2.31 **(Generator Suite)**
A function $\gamma : Variable \longrightarrow (A \longrightarrow \mathbf{P}A)$ is a *generator suite* iff
$\text{dom } \gamma = Variable \wedge \forall v \in Variable.\ Ord(v) = i \Rightarrow \gamma(v)$ is a level *i* generator.

It is convenient to refer to the function gained by composing the generators contained in a generator suite. This composite generator, which I call γ^*, yields the complete set of assignments to test given a set of values to consider.

Definition 2.32 **(Composite Generator)**
A *composite generator* $\gamma^* : \mathbf{P}Value \longrightarrow \mathbf{P}A_N$ is defined using a generator suite γ as
$$\gamma^*(\upsilon) = G_N(G_{N-1}(G_{N-2}(\ldots G_2(G_1(A_0,\upsilon),\upsilon)\ldots),\upsilon),\upsilon)$$
where $G_i(Q,\upsilon) = \bigcup_{q \in Q} g_i(q,\upsilon)$ and $g_i = \gamma(v_i)$.

Selective enumeration is the result of testing each full assignment generated by a composite generator, keeping only the satisfying assignments.

Definition 2.33 **(Selective Enumeration)**

Selective enumeration for the generator suite γ is the procedure that computes the function $\omega^*(\phi,\upsilon)$ for a formula ϕ and a set of values υ, where

$$\omega^*(\phi,\upsilon) = \{\ \alpha \in A_N \mid \alpha \in \gamma^*(\upsilon) \wedge M[\phi,\alpha,\upsilon] = \textsc{true}\ \}.$$

Lemma 2.9 Selective enumeration is a search.

Proof: Obvious.

If every generator in a generator suite is sound, then no satisfying equivalence class can be orphaned. The result of a composite generator containing only sound generators will therefore include at least one assignment for each satisfying equivalence class. Therefore, selective enumeration is sound if the generators used are sound.

Theorem 2.10 For any formula ϕ, the selective enumeration search $\omega^*(\phi,\upsilon)$ for the generator suite γ is sound for ϕ and υ if every generator in γ is sound for some duplication $\approx_d$.

Proof: By induction on level.

The induction hypothesis states that, at any level, for any satisfying assignment, the set of assignments generated by the composite generator up to that level will include the prefix of an assignment equivalent to the satisfying assignment.

$$\forall\alpha \in \mathrm{A}_N . M[\phi,\alpha,\upsilon] = \textsc{true} \wedge \mathrm{ran}\ \alpha \subseteq \upsilon \Rightarrow$$
$$\exists\alpha' \in A_N .\ \alpha \approx_d \alpha' \wedge Var_i \lhd \alpha' \in \mathrm{G}_i(\mathrm{G}_{i-1}(\ldots\ \mathrm{G}_1(A_0,\upsilon)\ \ldots\ ),\ \upsilon),\upsilon)$$

where $\mathrm{G}_i(\mathrm{X},\upsilon) = \bigcup_{x \in X} \gamma(v_i)(x,\upsilon)$

For the base case, where $i = 1$, the hypothesis to demonstrate is

$$\forall\alpha \in \mathrm{A}_N . M[\phi,\alpha,\upsilon] = \textsc{true} \wedge \mathrm{ran}\ \alpha \subseteq \upsilon \Rightarrow$$
$$\exists\alpha' \in A_N .\ \alpha \approx_d \alpha' \wedge Var_i \lhd \alpha' \in \mathrm{G}_1(A_0,\upsilon)$$

Because $A_0 = \{\ \emptyset\ \}$, $\mathrm{G}_1(A_0,\upsilon) = \gamma(v_1)(\emptyset,\upsilon)$.

Because $Var_0 \lhd a = \emptyset$, $\mathrm{G}_1(A_0,\upsilon) = \gamma(v_1)(Var_{i-1} \lhd \alpha,\upsilon)$.

Because $\gamma(v_1)$ is sound, the hypothesis holds directly from Lemma 2.7.

For the inductive case, we know that

$$\exists\alpha' \in A_N .\ a \approx_d \alpha' \wedge Var_{i-1} \lhd \alpha' \in \mathrm{G}_{i-1}(\ldots\ \mathrm{G}_1(A_0,\upsilon)\ \ldots\ ),\ \upsilon).$$

Therefore, G_i will include the results of $\gamma(v_i)(Var_{i-1} \lhd \alpha',\upsilon)$.

By Lemma 2.7, therefore, $\exists\alpha'' \in A_N .\ \alpha \approx_d \alpha'' \wedge Var_i \lhd \alpha'' \in \mathrm{G}_i(\mathrm{G}_{i-1}(\ldots\ \mathrm{G}_1(A_0,\upsilon)\ \ldots\ ),\ \upsilon),\upsilon)$.

Therefore, if there exists a full assignment α such that $M[\phi,\alpha,\upsilon] = \textsc{true} \wedge \mathrm{ran}\ \alpha \subseteq \upsilon$, then $\gamma^*(\phi,\upsilon)$ will contain at least one assignment α'.

Because $\alpha \approx_d \alpha'$ and $M[\phi,\alpha,\upsilon] = \textsc{true}$, $M[\phi,\alpha',\upsilon] = \textsc{true}$ and the search is sound.

2.6 Reductions and Efficiency

The goal of selective enumeration is to reduce the cost of searching for a solution. There are two primary costs involved in the search: generating assignments and testing those assignments against the formula. Selective enumeration attempts to reduce the number of assignments that are generated, and thus the number of assignments that are tested. For any given formula and set of values, the total cost of the search is (approximately) linearly related to the number of assignments (partial or full) that are generated.

The number of assignments generated during a selective-enumeration search depends both on the duplications chosen and on the generators implemented to remove the duplicates. A selective-enumeration technique is the combination of a duplication and a generator suite implementing that duplication. I therefore introduce measures of both factors in this section.

Because at most one full assignment from each equivalence class needs to be generated in a sound search, the fewer equivalence classes in the duplication, the fewer full assignments that need to be generated. I can thus compare the effect of two different duplications on the cost of a search by comparing the number of equivalence classes given by those duplications. The ratio of the number of equivalence classes gives a measure of the relative effectiveness of two duplications at reducing the cost of the search. Rather than comparing each pair of duplications, I instead compare each duplication to the duplication underlying exhaustive-enumeration search, given by $\approx_\varnothing$ in (2.3). This ratio, called the *reduction* by the duplication, is a number greater than or equal to one, with a larger number representing a more substantial reduction in the cost of the search. The reduction is equal to the mean size of the equivalence classes defined by the duplication.

Definition 2.34 **(Reduction)**

The reduction of a duplication $\approx_d$ for a formula ϕ and a set of values υ is

$$R(\approx_d,\phi,\upsilon) = \frac{|\{\, \alpha \in A_N \mid \operatorname{ran} \alpha \subseteq \upsilon \,\}|}{|\approx_d/\upsilon\,|}$$

where $|\approx_d/\upsilon\,|$ is the number of equivalence classes in $\approx_d$ including any assignments containing only values from υ.

The search does not need to generate exactly one assignment per satisfying equivalence class to be sound, it only needs to generate *at least* one assignment per equivalence class. For some duplications, including the ideal duplication $\approx_M$ defined in (2.4), perfect generators are computationally infeasible. For other duplications, including the isomorph duplication defined in Chapter 4, a perfect generator that generates exactly one assignment per equivalence class is computationally significantly more expensive than a conservatively approximate generator. Therefore, to fully compare duplications, the generator suite must also be considered.

The *efficiency* of a generator suite is the fraction of assignments generated that are necessary for each generator to be sound. A perfect generator suite has an efficiency of one — every assignment generated is necessary for a sound search. An inefficient generator will have an efficiency of less than one.

Definition 2.35 **(Efficiency)**

The efficiency of a generator suite γ for a duplication $\approx_d$, a formula ϕ, and a set of values υ is

$$E(\gamma,\approx_d,\phi,\upsilon) = \frac{|\approx_d/\upsilon\,|}{|\gamma^*(\upsilon)|}$$

where $|\approx_d/\upsilon\,|$ is the number of equivalence classes in $\approx_d$ including any assignments containing only values from υ.

The total cost of the search using generator suite γ, a duplication $\approx_d$, and a set of values υ is the cost of the exhaustive-enumeration search divided by the product of $R(\approx_d,\phi,\upsilon)$ and $E(\gamma,\approx_d,\phi,\upsilon)$. The cost of the exhaustive-enumeration search is proportional to the product of the number of values in the intersection of υ with *Typing*(v_i) for each variable v_i.

It is more useful, however, to compare individual generators than to compare entire generator suites. The efficiency of a single generator is defined in the same way as the efficiency of a generator suite, the fraction of assignments generated that are necessary for soundness.

To help compute this function, I define an extension of a set of partial assignments into a duplication as a set of full assignments that contains only assignments that are derivable from an element in the set of partial assignments and is unique with regards to the duplication.

Definition 2.36 **(*ext*)**

The function $ext(Q,\approx_d) : \mathsf{P}A_i \times Duplication \longrightarrow \mathsf{P}A_N$ is defined by the recursive procedure

```
if Q is empty
        $ext(Q,\approx_d) = \varnothing$
else
        for some $\alpha \in Q$
        $ext(Q,\approx_d) = ext(Q \setminus \{\alpha\},\approx_d) \cup$
                $\{ \alpha' : A_N \mid Var_i \lhd \alpha' = \alpha \wedge \forall\alpha'' \in ext(Q \setminus \{\alpha\},\approx_d).\ \neg\ \alpha' \approx_d \alpha'' \}$
```

The extension of a set of assignments generated by a sound generator includes exactly one assignment in each equivalence class. Any assignment generated that is not the prefix of one of the assignments is an extraneous assignment. The efficiency of a generator is computed as the ratio of the size of the extension of the result of the generator projected back to level *i* to the size of the actual result of the generator. Because the efficiency of a generator may vary with different initial partial assignments, I average the efficiency across all possible initial partial assignments.

Definition 2.37 **(Efficiency of a generator)**

The efficiency of a level *i* generator g for a duplication $\approx_d$, a formula ϕ, and a set of values υ is

$$E(g,\approx_d,\phi,\upsilon) = \frac{\sum\limits_{\alpha \in A_{i-1}} \left| \{ Var_i \lhd \alpha' \mid \alpha' \in ext(g(\alpha,\upsilon),\approx_d) \} \right|}{\sum\limits_{\alpha \in A_{i-1}} \left| g(\alpha,\upsilon) \right|}$$

If the efficiency of a generator were independent of the initial partial assignment, the product of the efficiencies of the individual generators in a generator suite would be the efficiency of the generator suite. In general, however, the input does affect the efficiency of the generator, although this appears only as a secondary effect in practice. The product of the efficiencies of each generator in a generator suite is therefore a reasonable approximation of the efficiency of the generator suite.

In the following two chapters I give heuristic computations to approximate the reduction of the two classes of duplications used by Ladybug and to approximate the efficiency of the generators implemented in Ladybug. In Chapter 6, I validate these approximations with the empirical results of checking specifications using Ladybug.

Chapter 3

Partial Assignment Duplication

This chapter describes partial-assignment duplications and how they are exploited by Ladybug. A partial-assignment duplication, like other duplications, allows a search to ignore assignments that "duplicate" other assignments that are generated by the search.

Considering two full assignments partial-assignment duplicates depends on two factors: a *filter formula* and a common partial assignment. The filter formula must follow from the formula being solved and the common partial assignment must fail to satisfy the filter formula. This failure guarantees that any full assignment extending the partial assignment will not satisfy the formula being solved by the search. Therefore the two full assignments that extend the common partial assignment must give the same interpretation to the formula being solved.

Ladybug uses three different techniques to exploit this pairing of a filter formula with partial assignments: short circuiting, derived variable elimination, and bounded generation. Each of these techniques has different performance implications and is enabled by a different class of filter formulae.

Short circuiting is the most broadly applicable of the techniques. It uses an underlying generator to generate a set of partial assignments. Short circuiting then filters each assignment generated through the filter formula, dropping any partial assignments that fail to satisfy the filter formula. Although short circuiting can be used for any filter formula that is fully bound by the partial assignments generated, it is the least efficient means of trimming the search tree, requiring many partial assignments to be generated and subsequently thrown away.

Derived variable elimination, on the other hand, is the most efficient means of generating the search tree, but has the most limited set of possible filter formulae. For a level *i* derived variable generator, there must be only a single value that can be bound to the i^{th} variable to satisfy the filter formula for any partial assignment from A_{i-1}. For Ladybug, only formulae that equate a variable to an expression using only the first *i-1* variables enable a level *i* derived variable generator.

Bounded generation occupies a middle ground between the efficient, but narrowly applicable approach of derived variable elimination and the inefficient, but broadly applicable approach of short circuiting. Like short circuiting, bounded generation uses an underlying generator to generate assignments. Unlike short circuiting, however, bounded generation restricts the set of values considered by the underlying generator, resulting in a smaller set of assignments being generated. The bounded-generation generator projects each of these assignments into assignments that are guaranteed to satisfy the filter formula. In this manner, bounded generation never generates assignments that are immediately discarded and is therefore much more efficient than short circuiting. Although bounded generation draws upon a more limited class of formulae than allowed by short circuiting, the class of formulae supporting bounded generation is much larger than the equalities required by derived variable elimination.

The first section of this chapter introduces another NP specification and solves a formula based on that specification to provide an example of how partial-assignment duplications can be exploited during a search. The second section formally defines a partial-assignment duplication and each form of partial-assignment duplication generator. The third section explores the workings of bounded generation in more detail. The fourth section describes the reduction available from a partial-assignment duplication. The final section places these techniques in the context of similar search reduction techniques developed by other researchers.

3.1 Example Search

This section informally considers how partial-assignment duplications can reduce the size of the search tree. I introduce a new specification to illustrate reducing the search tree.

Figure 3.1 lists a simple specification of the Macintosh desktop, including aliases and the trash can. Everything on the desktop is an object (an element of the given type OBJ). The desktop objects are partitioned into files and folders. Two distinct elements of folders are specially distinguished, the trash can, denoted by the variable trash, and the hard drive, denoted by the variable drive.The function dir relates each object to the folder that contains it (if any). Any object except the ones denoted by trash and drive can be contained in a folder. Some files are aliases (contained in the set aliases), and provide links to other objects (as denoted by the function links). This specification simplifies the system being modeled by disallowing aliases to link to other aliases. This restriction also disallows cyclic links functions. The set trashed consists of all objects that are directly or indirectly contained in trash.

Because the variables files, folders, drive, and trash are marked as const, their bindings cannot change across operations. This restriction means that the primed variables have the same binding as the unprimed variables. For example, in any operation based on Finder, files = files'. As a convenience, the resultant formula includes only the unprimed variables for these constant variables, replacing the primed variables with the base equivalents throughout.

Figure 3.1 specifies a single operation, Move. The operation Move reparents an object (given by the parameter x) from one folder to another (given by the parameter to). The preconditions not x = to and not x in dir+.{to} disallow moving a folder into itself or one of its (direct or indirect) children. The Move operation also indicates how the dir and links relations are modified by the operation. Without aliases, the update to dir would be simply dir' = ((OBJ\{x}) <: dir) U { x-> {to}}. The additional complexity inserts x into the folder for which to is an alias, if to is an alias.

The claim TrashingWorks states that moving an object into the trash can (or a folder contained in the trash can) should result in the object being considered trashed. Checking this claim requires finding a solution to the formula[1]

```
(3.1)        { drive } <= folders \ dom dir and { trash } <= folders \ dom dir and
             { drive } <= folders \ dom dir' and { trash } <= folders \ dom dir' and
             ran dir <= folders and ran dir' <= folders and
             trashed = dir~+. {trash } and trashed' = dir'~+. {trash } and
             not drive in trashed U { trash } and not drive in trashed' U { trash } and
```

[1] To improve readability, I have dropped several unnecessary parentheses required by the formula language grammar.

```
[OBJ]              /* Everything is an object */

/* Constrain what a legal desktop can look like */
Finder = [
 const files, folders: set OBJ     /* files and folders partition the objects */
 const drive, trash: OBJ           /* drive and trash are two fixed, special folders */
 dir, links: OBJ -> OBJ            /* dir relates objects to their enclosing folder,
                                     links relates an alias to its underlying object */
 trashed, aliases: set OBJ/        /* any object that has been thrown away is in trashed,
                                      aliases includes any object that is an alias */

|
 {drive, trash} <= folders \ dom dir  /* drive and trash are both folders and
                                                      not contained in any folder */
 ran dir <= folders                /* only folders can contain other objects */
 trashed = dir~+.{trash}           /* transitively contained in the trash is trashed */
 not drive in trashed U {trash}    /* the drive can never be in the trash */

 aliases <= files                  /* only files can be aliases */
 aliases = dom links               /* only aliases are links */
 aliases & ran links = {}          /* aliases are never liked to */
 links+ & Id = {}                  /* no cycles are allowed in the links */

 files & folders = {}              /* files and folders partition OBJ */
 files U folders = OBJ
]
/* Move an object (x) to inside another object (to) */
Move (x, to: OBJ) = [
 Finder
|
 not x = to                        /* The object  moved cannot already contain the target */
 not x in dir+.{to}

    /* update dir to indicate x is now in to (or what to refers to) */
 dir' = ((OBJ\{x}) <: dir) U { x-> (links U ((OBJ\aliases) <: Id)).{to} }
 links' = links                    /* links is unchanged */
]
/* check that this guarantees that trashing something puts it in the trash */
TrashingWorks (x, to: OBJ) :: [ Finder |
        Move (x, to) and to in trashed U {trash} => x in trashed' ]
```

Figure 3.1. A simple specification of the Macintosh desktop interface to the trash, including a simplified version of aliases.

```
aliases <= files and aliases' <= files and
aliases = dom links and aliases' = dom links' and
aliases & ran links = {} and aliases' & ran links' = {} and
links+ & Id = {} and links'+ & Id = {} and
files & folders = {} and files U folders = Un and
not x = to and not x in dir+.{to} and links' = links and
dir' = ((Un\{x}) <: dir) U { x-> (links U ((Un\aliases) <: Id)).{to} } and
to in trashed U {trash} and not x in trashed'
```

This formula involves twelve variables. For this example search, I assume the ordering

Ord = <links, links', aliases, aliases', folders, trash, drive, files, to, dir, trashed, x, dir', trashed'>

In this example, I define a universe of four elements, with OBJ=$\{ o_0, o_1, o_2, o_3 \}$.

Ladybug includes a mechanism for discovering potential filter formulae. This mechanism, which is fully described in Chapter 5, starts with the conjuncts of the formula being solved and adds any atomic formulae that it can discover that are implied by these formulae. For (3.1), the candidate formulae discovered by Ladybug include

```
{ drive } <= folders \ dom dir
{ trash } <= folders \ dom dir
{ drive } <= folders \ dom dir'
{ trash } <= folders \ dom dir'
ran dir <= folders
ran dir' <= folders
trashed = dir~+. {trash }
trashed' = dir'~+. {trash }
not drive in trashed U { trash }
not drive in trashed' U { trash }
aliases <= files
aliases' <= files
aliases = dom links
aliases' = dom links'
aliases & ran links = {}
aliases' & ran links' = {}
links+ & Id = {}
 links'+ & Id = {}
files & folders = {}
files U folders = Un
not x = to
not x in dir+.{to}
links' = links
dir' = ((Un\{x}) <: dir) U { x-> (links U ((Un\aliases) <: Id)).{to} }
to in trashed U {trash}
not x in trashed'
trash in folders
not drive in {trash}
drive in folders
Un \ folders <= files
files <= Un \ folders
not drive in dom dir
not trash in dom dir
```

In considering the possible choices of values for the first variable, links, all possible functions must be generated and bound to links, totaling 625 distinct assignments[2]. One of the formulae above, links+ & Id = {}, involves only the variable links. By testing each of the 625 assignments against this formula, 500 can be discarded as not satisfying the acyclic requirement, leaving 125 partial assignments at level 1. For example, the assignment binding the function $\{ o_0 \mapsto o_0 \}$ to links yields an assignment that does not satisfy this formula. To progress in this search example, I choose to bind $\{ o_0 \mapsto o_1 \}$ to links.

The next variable, links', must be bound to the same value as links, as required by the candidate formula links' = links. Rather than generating all 625 possible functions and testing each one for equality to the one chosen for links, the search will simply bind the same value to links' as was

[2] Only 155 of these functions are isomorphically distinct. As I shall demonstrate in Chapter 4, the techniques demonstrated in this chapter can be combined with isomorph elimination to further reduce the cost of the search.

already bound to links. This direct assignment is an example of a derived variable.

The third variable, aliases, can also be derived, using the filter formula aliases = dom links. Therefore, the assignment constructed thus far in the search is

$$\{\ \text{links} = \{\ o_0 \mapsto o_1\ \},\ \text{links'} = \{\ o_0 \mapsto o_1\ \},\ \text{aliases} = \{\ o_0\ \}\ \}.$$

The formula aliases & ran links = {}, which is one of the candidate formulae implied by (3.1), is fully bound by this assignment and can be used to further short circuit the search. Although the assignment chosen for this example search path satisfies this formula, many (84 to be exact) of the 125 level 1 partial assignments generated will be pruned here. The search binds the variable aliases' to the same value as aliases, using equivalent primed formulae. All remaining 41 partial assignments generated at level 3 can be extended to level 4.

All 16 possible sets must be considered for folders, as all candidate formulae that involve folders also includes a variable not yet enumerated in the search. For this example search path, I choose the partial assignment

$$\{\ \text{links} = \{\ o_0 \mapsto o_1\ \},\ \text{links'} = \{\ o_0 \mapsto o_1\ \},\ \text{aliases} = \{\ o_0\ \},\ \text{aliases'} = \{\ o_0\ \},\\ \text{folders} = \{\ o_1,\ o_2\ \}\ \}.$$

The current implementation of filter formula discovery in Ladybug is limited. An improved formula discovery process could reduce the number of level 5 assignments considered. The formula aliases & folders = {} is implied by the combination of aliases <= files and files & folders = {}, both of which are candidate formulae implied by (3.1). This missing formula allows eight assignments (all sets including o_0) to be generated along this path, only to be discovered as the basis of dead end paths later in the search.

The search could choose to generate assignments with each of the four objects bound to trash, the next variable in the search. Short circuiting would then test each of these four assignments against the candidate formula fully bound by these assignments, trash in folders. However, this formula also enables the more efficient bounded generation, which Ladybug always chooses over short circuiting when available. With bounded generation, only the two elements in the set currently bound to folders, o_1 and o_2, are considered. This restriction guarantees that the candidate formula is satisfied and no short-circuiting tests are necessary. For this example search path, I choose to bind the element o_2 to trash, yielding the assignment

$$\{\ \text{links} = \{\ o_0 \mapsto o_1\ \},\ \text{links'} = \{\ o_0 \mapsto o_1\ \},\ \text{aliases} = \{\ o_0\ \},\ \text{aliases'} = \{\ o_0\ \},\\ \text{folders} = \{\ o_1,\ o_2\ \},\ \text{trash} = o_2\ \}.$$

The other distinguished folder, drive, is the next variable to be bound. Two candidate formulae involve only drive and variables already bound: not drive in { trash } and drive in folders. Each of these formulae enable bounded generation, jointly requiring that drive be an element of folders \ { trash }. For the current search path, this restriction means that only the element o_1 will be considered for drive. Although this restriction limits drive to a single possible value, drive is not a derived variable, as other partial assignments will leave more than one value to consider.

The search also uses two formulae to limit the possible values for the next variable, files, to a single choice. The limitations in the filter formulae discovery mechanism prevents files from being a derived variable. Bounded generation, however, will always limit the number of values generated for files to exactly one: the complement of folders. The candidate formula aliases <= files is also fully bound by a level 8 assignment and enables further bounded-generation opportunities. Whenever the earlier unrecognized formula aliases & folders = {} is satisfied, there is exactly one set to be considered. For the example search path being considered, the partial assignment construct thus far is

$$\begin{aligned}\{ &\text{links} = \{ o_0 \mapsto o_1 \}, \text{links'} = \{ o_0 \mapsto o_1 \}, \text{aliases} = \{ o_0 \}, \text{aliases'} = \{ o_0 \},\\ &\text{folders} = \{ o_1, o_2 \}, \text{trash} = o_2, \text{drive} = o_1, \text{files} = \{ o_0, o_3 \} \}.\end{aligned}$$

The next variable to be bound is to. Every candidate formulae involving to also involves variables not yet bound, so all 4 possible values of to must be considered. For the example search path, I choose to bind the element o_0 to the variable to.

The next variable, dir, is a function that is supported by five candidate formulae:

```
not drive in dom dir
not trash in dom dir
ran dir <= folders
{ drive } <= folders \ dom dir
{ trash } <= folders \ dom dir.
```

The first three of these formulae enable bounded generation, meaning that the domain and range of any values to be considered for dir are limited (e.g. dir $\in \mathbf{P}(\{ o_0, o_3 \} \times \{ o_1, o_2 \})$). From the nine functions satisfying this constraint, I choose the value $\{ o_0 \mapsto o_2 \}$ to bind to the variable dir in this example. The fourth and fifth candidate formulae are used to short circuit the search. This short circuiting will gain no reduction, as these formulae are implied by other filter formulae already considered in the search.

The next variable, trashed, can be derived from the formula trashed = dir~+.{trash}. The search therefore binds the value $\{ o_0 \}$ to trashed. Two more formulae, not drive in trashed U { trash } and to in trashed U { trash }, can prune some assignments generated thus far, but allow the assignment constructed by the example search to continue. The assignment constructed thus far is

$$\begin{aligned}\{ &\text{links} = \{ o_0 \mapsto o_1 \}, \text{links'} = \{ o_0 \mapsto o_1 \}, \text{aliases} = \{ o_0 \}, \text{aliases'} = \{ o_0 \},\\ &\text{folders} = \{ o_1, o_2 \}, \text{trash} = o_2, \text{drive} = o_1, \text{files} = \{ o_0, o_3 \},\\ &\text{to} = o_0, \text{dir} = \{ o_0 \mapsto o_2 \}, \text{trashed} = \{ o_0 \} \}.\end{aligned}$$

The next variable, x, is the last non-derived variable considered in the search. Two candidate formulae enable bounded generation for x: not x = to and not x in dir+.{to}. These formulae require that the value of x be drawn from Un \ ({to} U dir+.{to}). In the example search constructed thus far, this allows any element other than o_0 to be considered for x. For this example search path, I choose the value o_3 for x.

The variable dir' can be derived from the assignment constructed thus far, based on the formula dir' = ((Un\{x}) <: dir) U { x-> (links U ((Un\aliases) <: Id)).{to} }. For the example assignment constructed thus far, this requires dir' to be bound to $\{ o_0 \mapsto o_2, o_3 \mapsto o_1 \}$. Three candidate formulae, ran dir' <= folders, { drive } <= folders \ dir', and { trash } <= folders \ dir', enable further short circuiting of the search tree.

Finally, the variable trashed' must be bound to the value of dir'+.{ trash }, based on the candidate formula trashed' = dir'+.{ trash }. The resultant full assignment only needs to be checked against the formula not drive in trashed' U { trash }. The following full assignment constructed by the example search satisfies this formula, thus providing a counterexample to the claim TrashingWorks.

$$\begin{aligned}\{ &\text{links} = \{ o_0 \mapsto o_1 \}, \text{links'} = \{ o_0 \mapsto o_1 \}, \text{aliases} = \{ o_0 \}, \text{aliases'} = \{ o_0 \},\\ &\text{folders} = \{ o_1, o_2 \}, \text{trash} = o_2, \text{drive} = o_1, \text{files} = \{ o_0, o_3 \},\\ &\text{to} = o_0, \text{dir} = \{ o_0 \mapsto o_2 \}, \text{trashed} = \{ o_0 \}, \text{x} = o_3,\\ &\text{dir'} = \{ o_0 \mapsto o_2, o_3 \mapsto o_1 \}, \text{trashed'} = \{ o_0 \} \}.\end{aligned}$$

In this counterexample, an object is moved into an alias for a folder. The alias is in the trash, but the underlying folder is not. In the model, this action does not result in the object being in the trash. In the actual Macintosh desktop, this action results in an alert disallowing the action and recommending that the user remove the alias from the trash.

The example search path I demonstrated required no backtracking to discover this counterexample. Ladybug, using a simple, smallest-first value selection process, requires 9,633 different paths before locating this counterexample. Although some backtracking is required, this number still represents a huge reduction from the total search tree, which contains 6.55×10^{20} total search paths. Even finding all 408 counterexamples requires only 36,288 distinct search paths to be considered. By integrating isomorph elimination, the number of paths considered drops to only 92 to find both isomorphically distinct counterexamples. Each of these searches requires less than a second to complete.

3.2 Formal Definitions

This section formally defines the three techniques that exploit a partial-assignment duplication. As a reminder, I re-introduce the definition of a partial-assignment duplication given by Definition 2.26.

Definition 2.26 **(Partial Assignment Duplication)**

An equivalence relation $\approx_{\rho(\phi')}$ is a *partial-assignment duplication* of the filter formula ϕ' for a formula ϕ and a set of values υ iff

$\phi \vDash \phi' \wedge \forall \alpha, \alpha' \in A_N.\ (\alpha \approx_{\rho(\phi')} \alpha' \Leftrightarrow (\alpha = \alpha' \vee (M[\phi',\alpha,\upsilon] = \text{FALSE} \wedge M[\phi',\alpha',\upsilon] = \text{FALSE})))$.

The key to a partial-assignment duplication is the filter formula ϕ'. Two distinct full assignments are duplicates only if neither satisfies ϕ'. Each assignment satisfying ϕ' forms its own equivalence class in $\approx_{\rho(\phi')}$, with all other full assignments combining into a single equivalence class.

To be useful to a level *i* generator implementing any of these approaches, the variables of the filter formula must be limited to a subset of Var_i. Each technique described in this section places additional constraints on this filter formula.

Short circuiting is independent of the problem domain. A short circuiting generator uses an underlying generator to generate a set of assignments, and then yields the subset of those assignments that satisfies the filter formula.

Definition 3.1 **(Short Circuiting Generator)**

A level *i* generator $g : A_{i-1} \times \mathbb{P}Value \longrightarrow \mathbb{P}A_i$ with an underlying level *i* generator g' and a filter formula ϕ' is a level *i* short-circuiting generator for a formula ϕ and a set of values υ iff

$\phi \vDash \phi' \wedge Var(\phi') \subseteq Var_i \wedge g(\alpha,\upsilon) = \{ a \mid a \in g'(\alpha,\upsilon) \wedge M[\phi',a,\upsilon] = \text{TRUE} \}$.

A short circuiting generator is sound if the underlying generator is sound because short circuiting only excludes partial assignments with no satisfying extensions.

Theorem 3.1 A level *i* short-circuiting generator for a formula ϕ and a set of values υ with an underlying level *i* generator g' and a filter formula ϕ' is sound for any duplication $\approx$ if g' is sound for $\approx$.

Proof: By Definition 2.29, g is sound for $\approx$ iff

$\forall \upsilon \subseteq Value.\ \forall \alpha \in A_N.\ (M[\phi,\alpha,\upsilon] = \text{TRUE} \wedge \operatorname{ran} \alpha \subseteq \upsilon) \Rightarrow$

$$\exists\alpha' \in A_N.\ \mathit{Var}_i \lhd \alpha' \in g(\mathit{Var}_{i-1} \lhd \alpha,\upsilon) \wedge \alpha \approx \alpha' \wedge \operatorname{ran}\alpha' \subseteq \upsilon$$

Let $\alpha \in A_N$ such that $M[\phi,\alpha,\upsilon] = \text{TRUE} \wedge \operatorname{ran}\alpha \subseteq \upsilon$.

Because g' is sound for $\approx$,

$$\exists\alpha' \in A_N.\ \mathit{Var}_i \lhd \alpha' \in g'(\mathit{Var}_{i-1} \lhd \alpha,\upsilon) \wedge \alpha \approx \alpha' \wedge \operatorname{ran}\alpha' \subseteq \upsilon.$$

Because $\approx$ is a duplication and $\alpha \approx \alpha'$ and $M[\phi,\alpha,\upsilon] = \text{TRUE}$,

$$M[\phi,\alpha',\upsilon] = \text{TRUE}.$$

Therefore, because $\phi \vDash \phi'$, $M[\phi',\alpha',\upsilon] = \text{TRUE}$.

Therefore, because $\mathit{Var}(\phi') \subseteq \mathit{Var}_i$, by Lemma 2.2, $M[\phi', \mathit{Var}_i \lhd \alpha',\upsilon] = \text{TRUE}$.

Therefore, by Definition 3.1, $\mathit{Var}_i \lhd \alpha' \in g(\mathit{Var}_{i-1} \lhd \alpha,\upsilon)$. ■

A short-circuiting generator reduces the cost of the search by pruning some uninteresting paths from the tree, but at a significant cost. For the last level of the tree, this cost is identical to the savings. Therefore, short circuiting is not appropriate for use as a level *N* generator.

Unlike short circuiting, derived variables are appropriate, if enabled, at any level. Deriving a variable is enabled if a filter formula is only satisfied by at most one level *i* assignment that extends from each initial assignment. The generator returns the set containing only that single assignment. If no satisfying extensions exist, a short circuiting generator returns the empty set.

Definition 3.2 **(Derived-Variable Generator)**

A level *i* generator $g : A_{i-1} \times \mathbb{P}\mathit{Value} \rightarrow \mathbb{P}A_i$ with a filter formula ϕ' is a level *i* derived-variable generator for a formula ϕ and a set of values υ iff

$$\phi \vDash \phi' \wedge \mathit{Var}(\phi') \subseteq \mathit{Var}_i \wedge \forall\alpha \in A_{i-1}.$$
$$(\exists x \in \upsilon \cap \mathit{Typing}(v_i).(M[\phi',\alpha \cup \{ v_i \mapsto x \},\upsilon] = \text{TRUE} \wedge$$
$$\forall x' \in \upsilon \cap \mathit{Typing}(v_i) \setminus \{x\}.M[\phi',\alpha \cup \{ v_i \mapsto x' \},\upsilon] = \text{FALSE}) \wedge$$
$$g(\alpha,\upsilon) = \{ \alpha \cup \{ v_i \mapsto x \} \}) \vee$$
$$(\forall x \in \upsilon \cap \mathit{Typing}(v_i).M[\phi',\alpha \cup \{ v_i \mapsto x \},\upsilon] = \text{FALSE} \wedge$$
$$g(\alpha,\upsilon) = \varnothing))$$

where $v_i \in \mathit{Variable}$ and $\mathit{Ord}(v_i) = i$.

A derived-variable generator is sound when enabled because the filter formula ϕ' sufficiently constrains the problem to guarantee that the single assignment generated is the only possibly satisfying extension of the initial assignment.

Theorem 3.2 A level *i* derived-variable generator g for a formula ϕ and a set of values υ with a filter formula ϕ' is sound for any duplication $\approx$ if $\forall\alpha \in A_{i-1}$.

$$((\exists x \in \upsilon.(\forall\alpha' \in A_i.(\mathit{Var}_{i-1} \lhd \alpha' = \alpha \wedge M[\phi',\alpha',\upsilon] = \text{TRUE}) \Rightarrow \alpha'(v_i) = x)) \vee$$
$$(\forall x \in \upsilon \cap \mathit{Typing}(v_i).M[\phi',\alpha \cup \{ v_i \mapsto x \},\upsilon] = \text{FALSE}))$$

where $v_i \in \mathit{Variable}$ and $\mathit{Ord}(v_i) = i$..

Proof: Let $a \in A_N$ such that $M[\phi,a,\upsilon] = \text{TRUE} \wedge \operatorname{ran} a \subseteq \upsilon$.
If no such assignment exists, any result of the generator, including the empty set, would be sound.

Because $\phi \vDash \phi'$, $M[\phi',a,\upsilon] = \text{TRUE}$.

Therefore, because $\mathit{Var}(\phi') \subseteq \mathit{Var}_i$, $M[\phi', \mathit{Var}_i \lhd a,\upsilon] = \text{TRUE}$.

Therefore, there exists a value x and the first enabling condition is relevant.

Therefore, $g(\mathit{Var}_{i-1} \lhd a,\upsilon) = \{ \mathit{Var}_{i-1} \lhd a \cup \{ v_i \mapsto a(v_i) \} \}$.

Obviously, $\{ \mathit{Var}_{i-1} \lhd a \cup \{ v_i \mapsto a(v_i) \} \} = \mathit{Var}_i \lhd a$.

Therefore, $\mathit{Var}_i \lhd a \in g(\mathit{Var}_{i-1} \lhd a, \upsilon) \wedge a \approx a \wedge \mathrm{ran}\ \mathit{Var}_i \lhd a \subseteq \upsilon$ ■

Derived-variable generators give large reductions with little cost, an ideal combination. Unfortunately, relatively few variables in practice are constrained by an enabling filter formula.

Bounded generation is appropriate where the filter formula constrains the value to something less than all of υ, but more than a single value. Although the specific forms of constraints supported vary across problem domains and implementations, two classes of constraints are generally available.

The simplest constraints require that the value for the i^{th} variable be chosen from an easily described subset of the set of values υ (e.g. the powerset of a set). As an example, assume that t and s are elements of Var_{set} and ordered as $i\text{-}1^{th}$ and i^{th} variables respectively. The formula s <= t constrains s to be bound to a subset of the set bound to t in any assignment that satisfies this formula. A level *i* bounded-generation generator uses a reduced set of values equal to the powerset of t. Bounded generation passes this reduced set of values to an underlying generator to generate assignments. All assignments generated by this underlying generator will satisfy the formula s <= t.

Other constraints limit the value for the i^{th} variable to be an element of a set cleanly described as projections of an easily described subset of the set of values υ. An obvious example is the formula z in s, where s is a set and z is a scalar variable enumerated before s. A possible approach involves enumerating all sets that include the value of z. This set of values, however, is not the powerset of any set and therefore is more expensive to describe. This set is also not closed under permutation, causing an additional problem in the implementation.

To avoid these problems, bounded generation chooses a different set of values for the underlying generator: the set of all values not containing the value of z. This set is easily described as a powerset ($\mathbb{P}(U \backslash \{z\})$) and is closed under permutation and set difference. Bounded generation then uses a *projection function* to insert the value of z into each set bound to s in the assignments generated by the underlying generator. In general, the projection function adjusts the newly bound value in each assignment returned by the underlying generator to satisfy the filter formula.

To combine these two cases, I view the simpler constraints as a special case of the more complex constraints. All bounded-generation generators require four pieces: a filter formula, an underlying generator, a reduced set of values, and a projection function. The simpler constraints define the projection function to be the identity. Bounded generation restricts the values considered by the underlying generator to the reduced set of values. Bounded generation uses the projection function to map each assignment yielded by the underlying generator into one that satisfies the filter formula.

Definition 3.3 **(Bounded-Generation Generator)**

A level *i* generator $g : A_{i-1} \times \mathbb{P}\mathit{Value} \rightarrow \mathbb{P}A_i$ for a formula ϕ and a set of values υ with a filter formula ϕ', an underlying generator g', reduced set of values υ', and a projection function proj defined as

$\phi' : \mathit{Wff} .\ \phi \models \phi' \wedge\ \mathit{Var}(\phi') \subseteq \mathit{Var}_i$

$g' : A_{i-1} \times \mathbb{P}\mathit{Value} \rightarrow \mathbb{P}A_i$. g' is a level *i* generator

$\upsilon' : \mathbb{P}\mathit{Value} .\ \upsilon' \subseteq \upsilon$

$\mathrm{proj} : (\upsilon' \cap \mathit{Typing}(v_i)) \times A_{i-1} \times \mathbb{P}\mathit{Value} \rightarrow (\upsilon \cap \mathit{Typing}(v_i))$.

$\forall \alpha \in A_{i-1}.\ \forall x \in \upsilon \cap \mathit{Typing}(v_i).$

$M[\phi', \alpha \cup \{ v_i \mapsto x \}, \upsilon] = \mathrm{TRUE} \Rightarrow \exists x' \in \upsilon'.\ x = \mathrm{proj}(x', \alpha, \upsilon)$

is a level *i* bounded-generation generator iff

$$\forall \alpha \in A_{i-1}.\ g(\alpha,\upsilon) = \{\ \mathrm{proj}(\alpha(v_i),\alpha,\upsilon)\ \mid\ \alpha \in g'(\alpha,\upsilon')\ \}.$$

where $v_i \in \mathit{Variable}$ and $\mathit{Ord}(v_i) = i$.

Ideally, the next theorem would state that bounded generation is sound if the underlying generator is sound for some duplication $\approx$. Because the projection function may change the assignments generated by the underlying generator, it may transform the lone representative of some equivalence class of $\approx$ into an element of a separate equivalence class for $\approx$. In the previous example supported by the filter formula z in s, bounded generation uses the projection function to insert the value of z into the set bound to s in each assignment yielded by the underlying generator. This transformation maps an assignment that is not the prefix of any satisfying full assignment into one that may be the prefix of a satisfying full assignment. Therefore the underlying generator is not sound in the context of the full set of values.

The underlying generator must satisfy a different sense of soundness to guarantee that the bounded-generation generator is sound. *Limited soundness* for a generator is similar to soundness with two notable differences. The newly bound value must be from a reduced set of values and the results of the generator must cover the duplication only after each assignment in the result is projected through a projection function.

Definition 3.4 **(Limited Soundness)**

A level *i* generator g is *limited sound* for a duplication $\approx$, a formula ϕ, and a set of values υ under a reduced set of values υ' and a projection function $\mathrm{proj} : (\upsilon' \cap \mathit{Typing}(v_i)) \times A_{i-1} \times \mathbf{P}\,\mathit{Value} \rightarrow (\upsilon \cap \mathit{Typing}(v_i))$ iff

$$\begin{aligned}&\upsilon' \subseteq \upsilon \wedge \forall \alpha \in A_N.\ (M[\phi,\alpha,\upsilon] = \text{TRUE} \wedge \mathrm{ran}\ \alpha \subseteq \upsilon) \Rightarrow\\ &\qquad \exists \alpha' \in A_N.\ \alpha \approx \alpha' \wedge \mathrm{ran}\ \alpha' \subseteq \upsilon \wedge\\ &\qquad\qquad \exists \alpha'' \in g(\mathit{Var}_{i-1} \lhd \alpha,\upsilon').\ \mathit{Var}_{i-1} \lhd \alpha' = \mathit{Var}_{i-1} \lhd \alpha'' \wedge\\ &\qquad\qquad\qquad \alpha'(v_i) = \mathrm{proj}(\alpha''(v_i), \mathit{Var}_{i-1} \lhd \alpha,\upsilon)\end{aligned}$$

where $v_i \in \mathit{Variable}$ and $\mathit{Ord}(v_i) = i$.

If the reduced set of values is the inclusive subset and the projection function is the identity, the definition of limited soundness degenerates to the definition of soundness. All sound generators therefore are limited sound for the inclusive set of values and the identity function. Many generators that are sound are also limited sound under other subsets of values and projection functions. The exhaustive-enumeration generator *gx*, given in Definition 2.30, is limited sound for any reduced set of values which is projected by the projection function to include all values satisfying some formula implied by the formula being solved.

Lemma 3.3 The level *i* exhaustive-enumeration generator *gx* for a formula ϕ and a set of values υ is limited sound under a set of values υ' and a projection function proj if

$$\begin{aligned}&\exists \phi' \in \mathit{Wff}.\ \phi \vDash \phi' \wedge \mathit{Var}(\phi') \subseteq \mathit{Var}_i \wedge\\ &\qquad \forall \alpha \in A_{i-1}.\ \forall x \in \upsilon \cap \mathit{Typing}(v_i).\\ &\qquad\qquad M[\phi',\alpha \cup \{\ v_i \mapsto x\ \},\upsilon] = \text{TRUE} \Rightarrow \exists x' \in \upsilon'.\ x = \mathrm{proj}(x',\alpha,\upsilon)\end{aligned}$$

where $v_i \in \mathit{Variable}$ and $\mathit{Ord}(v_i) = i$.

Proof: Let $\alpha \in A_N$ such that $M[\phi,\alpha,\upsilon] = \text{TRUE} \wedge \mathrm{ran}\ \alpha \subseteq \upsilon$ and let $x \in \upsilon \cap \mathit{Typing}(v_i)$ such that $M[\phi', \mathit{Var}_{i-1} \lhd \alpha \cup \{\ v_i \mapsto x'\ \},\upsilon] = \text{TRUE}$

Therefore, $\exists x' \in \upsilon'.\ x = \mathrm{proj}(x', \mathit{Var}_{i-1} \lhd \alpha,\ \upsilon)$.

By the definition of *gx*, because $x' \in \upsilon'$,

$$(\mathit{Var}_{i\text{-}1} \lhd \alpha \cup \{ v_i \mapsto x' \}) \in gx(\mathit{Var}_{i\text{-}1} \lhd \alpha, \upsilon').$$

Therefore, $\exists \alpha' \in g(\mathit{Var}_{i\text{-}1} \lhd \alpha, \upsilon').\ \mathit{Var}_{i\text{-}1} \lhd \alpha = \mathit{Var}_{i\text{-}1} \lhd \alpha' \wedge$
$\alpha(v_i) = \mathrm{proj}(\alpha'(v_i), \mathit{Var}_{i\text{-}1} \lhd \alpha, \upsilon)$. ■

A level *i* bounded-generation generator is sound if the underlying generator is limited sound.

Theorem 3.4 A level *i* bounded-generation generator g for a formula ϕ and a set of values υ with the underlying level *i* generator g', set of values υ', projection function proj, and filter formula ϕ' is sound for a duplication $\approx$ if g' is limited sound for $\approx$ under the set of values υ' and the projection function proj.

Proof: Let $\alpha \in A_N$ such that $M[\phi, \alpha, \upsilon] = \text{TRUE} \wedge \mathrm{ran}\ \alpha \subseteq \upsilon$.

To prove soundness, it is necessary to show that,

$$\exists \alpha' \in A_N.\ \mathit{Var}_i \lhd \alpha' \in g(\mathit{Var}_{i\text{-}1} \lhd \alpha, \upsilon) \wedge \alpha \approx \alpha' \wedge \mathrm{ran}\ \alpha' \subseteq \upsilon$$

Because g' is limited sound for $\approx$ under υ' and proj,

$$\begin{aligned} &\exists \alpha' \in A_N.\ \alpha \approx \alpha' \wedge \mathrm{ran}\ \alpha' \subseteq \upsilon \wedge \\ &\quad \exists \alpha'' \in g'(\mathit{Var}_{i\text{-}1} \lhd \alpha, \upsilon').\ \mathit{Var}_{i\text{-}1} \lhd \alpha' = \mathit{Var}_{i\text{-}1} \lhd \alpha'' \wedge \\ &\qquad \alpha'(v_i) = \mathrm{proj}(\alpha''(v_i), \mathit{Var}_{i\text{-}1} \lhd \alpha, \upsilon). \end{aligned}$$

Therefore, $\exists \alpha' \in A_N.\ \alpha \approx \alpha' \wedge \mathrm{ran}\ \alpha' \subseteq \upsilon \wedge \mathit{Var}_i \lhd \alpha' \in g(\mathit{Var}_{i\text{-}1} \lhd \alpha, \upsilon)$. ■

Although not exploited by Ladybug, another form of duplication similar to partial-assignment duplication is also available for reducing the search space. A *disjunctive partial-assignment duplication* considers two full assignments to be equivalent iff they both contain a common partial assignment that satisfies a filter formula and that filter formula implies the formula being solved. Whereas a partial-assignment duplication groups assignments that are not satisfying by testing them against a conjunct of the formula being solved, a disjunctive partial-assignment duplication groups satisfying assignments that contain a common partial assignment that satisfies a disjunct of the formula being solved.

Definition 3.5 **(Disjunctive Partial Assignment Duplication)**
An equivalence relation $\approx_{\vee(\phi')}$ is a *disjunctive partial-assignment duplication* of the filter formula ϕ' for a formula ϕ and a set of values υ iff
$\phi' \models \phi \wedge\ \forall \alpha, \alpha' \in A_N.\ (\alpha \approx_{\vee(\phi')} \alpha' \Leftrightarrow (\alpha = \alpha' \vee (M[\phi', \alpha, \upsilon] = \text{TRUE} \wedge M[\phi', \alpha', \upsilon] = \text{TRUE})))$.

Unlike a partial-assignment duplication, a disjunctive partial-assignment duplication needs to retain one representative of each equivalence class. Therefore, the techniques used to exploit a partial-assignment duplication are not directly applicable to a disjunctive partial-assignment duplication. Ladybug ignores this duplication, as the default normalization it uses eliminates disjunctions from the formulae before solving the clauses using selective enumeration.

3.3 Bounded Generation

Bounded generation is the most interesting and the most complicated of the three techniques addressed in this chapter. This section explores the workings of bounded generation in detail.

The selection of a filter formula is the primary consideration in using bounded generation. The choice of a filter formula for short circuiting, because of its broad applicability, and for derived variable generation, because of its narrow applicability, is straightforward. Any filter formula using only the first *i* variables enables short circuiting. Only filter formulae equating the i^{th} variable to the value of a term using only the first *i-1* variables enables derived variable elimination. In

Ladybug, bounded generation can exploit filter formulae that use the first *i* variables and match any of the bounded-generation patterns. Table 3.1 shows the complete set of patterns used by Ladybug and the effect of these formulae on the reduced set of values and projection function.

For set variables, the applicable formulae fall into three categories: atomic formulae that restrict the value bound to a variable to be a subset of the value denoted by a set term, atomic formulae that restrict the value bound to a variable to be a superset of the value denoted by a set term, and atomic formulae that restrict the value bound to a set variable to be disjoint from the value denoted by a set term. In every case, the associated set term must include only variables from Var_{i-1}.

Filter Formula	**Reduced values**	**Projection function**
$v_{set} <= \tau$	$\mathbf{P}M_{set}[\tau,\alpha,\upsilon]$	$proj(X,\alpha,\upsilon) = X$
$\tau <= v_{set}$	$\mathbf{P}M_{set}[(Un \setminus \tau),\alpha,\upsilon]$	$proj(X,\alpha,\upsilon) = M_{set}[(X \cup \tau),\alpha,\upsilon]$
$(v_{set} \& \tau) = \{\}$	$\mathbf{P}M_{set}[(Un \setminus \tau),\alpha,\upsilon]$	$proj(X,\alpha,\upsilon) = X$
v_{scalar} in τ	$M_{set}[\tau,\alpha,\upsilon]$	$proj(x,\alpha,\upsilon) = x$
not v_{scalar} in τ	$M_{set}[(Un \setminus \tau),\alpha,\upsilon]$	$proj(x,\alpha,\upsilon) = x$
not $v_{scalar} = \tau$	$M_{set}[(Un \setminus \{\tau\}),\alpha,\upsilon]$	$proj(x,\alpha,\upsilon) = x$
dom $v_{rel} <= \tau$	$\mathbf{P}(M_{set}[\tau,\alpha,\upsilon] \times U)$	$proj(R,\alpha,\upsilon) = R$
ran $v_{rel} <= \tau$	$\mathbf{P}(U \times M_{set}[\tau,\alpha,\upsilon])$	$proj(R,\alpha,\upsilon) = R$
func v_{rel} and $\tau <= v_{rel}$	$\mathbf{P}(M_{set}[(Un \setminus dom\ \tau),\alpha,\upsilon] \times U)$	$proj(R,\alpha,\upsilon) = M_{rel}[(R \cup \tau),\alpha,\upsilon]$
func $(v_{rel}\sim)$ and $\tau <= v_{rel}$	$\mathbf{P}(M_{set}[U \times (Un \setminus ran\ \tau),\alpha,\upsilon])$	$proj(R,\alpha,\upsilon) = M_{rel}[(R \cup \tau),\alpha,\upsilon]$

Table 3.1: Forms of filter formulae supporting bounded generation. The first column gives a description of each filter formula that supports bounded generation. The second column indicates the values that can still be included in the reduced set of values. The third column gives the projection function required by each filter formula.

Bounded generation converts each of these cases into a reduced set of values and a projection function. For the first pattern, set subsetting, the reduced set of values is simply the powerset of the value bound to the set term and the projection function is the identity. For the second category, set supersetting, the reduced set of values is the powerset of the complement of the value denoted by the set term and the projection function yields the union of value denoted by the set term and the value bound to i^{th} variable. The disjoint case also uses the powerset of the complement of the value denoted by the set term as the reduced set of values, but uses the identity function as the projection function.

Scalar variables have three categories of available filter formulae: membership in a set, the negation of membership in a set, and negated equality to another scalar. Again, all filter terms must include only variables from Var_{i-1}. For the first case, membership, the reduced set of values is the set denoted by the set term. For the second case, negated membership, the reduced set of values is the complement of the set term. The third case, inequality, indicates that the reduced set of values is the universe of atomic elements minus the scalar denoted by the other scalar term. In all scalar cases, the projection function is the identity.

Rather than subsetting and supersetting the relations themselves, Ladybug generally considers subsetting only the domains and ranges of relation values. If the filter formula restricts the domain of the i^{th} relational variable to be a subset of a set term including only variables from Var_{i-1}, the reduced set of values includes all relations where the domain is a subset of the indicated set term. The range or both the domain and the range can be restricted similarly. For either of these cases, the projection function is the identity.

Bounded generation will subset the relation value itself only when the relational variable is constrained to be a function. The reduced set of values for this case includes only values in the complement of the domain of the value denoted by τ. The projection function in this case yields the union of the value denoted by τ and the relation bound to the i^{th} variable. Unlike other cases of bounded generation, the generator does not guarantee that the filter formula is satisfied. In particular, if the value denoted by τ is not a function, the newly generated relation will not be either. This case also applies to relations constrained to be injections, reducing the range rather than the domain.

If multiple atomic formulae support bounded generation for the i^{th} variable, the filter formula used is the conjunction of two formulae. The resultant reduced set of values is the intersection of the reduced sets of values from each atomic formula and the resultant projection functions is the composition of the projection function from each atomic formula[3].

I return to the Alloc example to demonstrate how bounded generation works. The set of formulae implied by (2.2) discovered by Ladybug includes

1) dom usage <= used
2) dom usage' <= used'
3) used <= used'
4) func usage and (used <: usage') <= usage
5) ran usage <= ran usage'
6) newAddr in used'
7) newAddr in used

For bounded generation alone, Ladybug chooses the variable ordering

$$Ord = < \text{used, used', newAddr, usage', usage} \}$$

Because all the formulae consider reference variables enumerated after used, no bounded generation is available for used. If Ladybug uses an exhaustive-enumeration generator for used, all possible sets will be generated.

The third formula (used <= used') enables the bounded generation of used'. All sets containing any element found in used are excluded from the set of reduced values. The value of used' in each assignment generated by the bounded-generation generator is the union of the value of used and the value bound to used' in the assignment generated by the underlying generator.

The last two formulae (newAddr in used', newAddr in used) support the bounded generation of newAddr. Without bounded generation, each element in Addr would need to be considered for newAddr. Instead, the reduced set of values passed to the underlying generator includes only the intersection of the values of used' and used.

[3] The order of composition is inconsequential as union, the only operation used in the projection functions, is both associative and commutative.

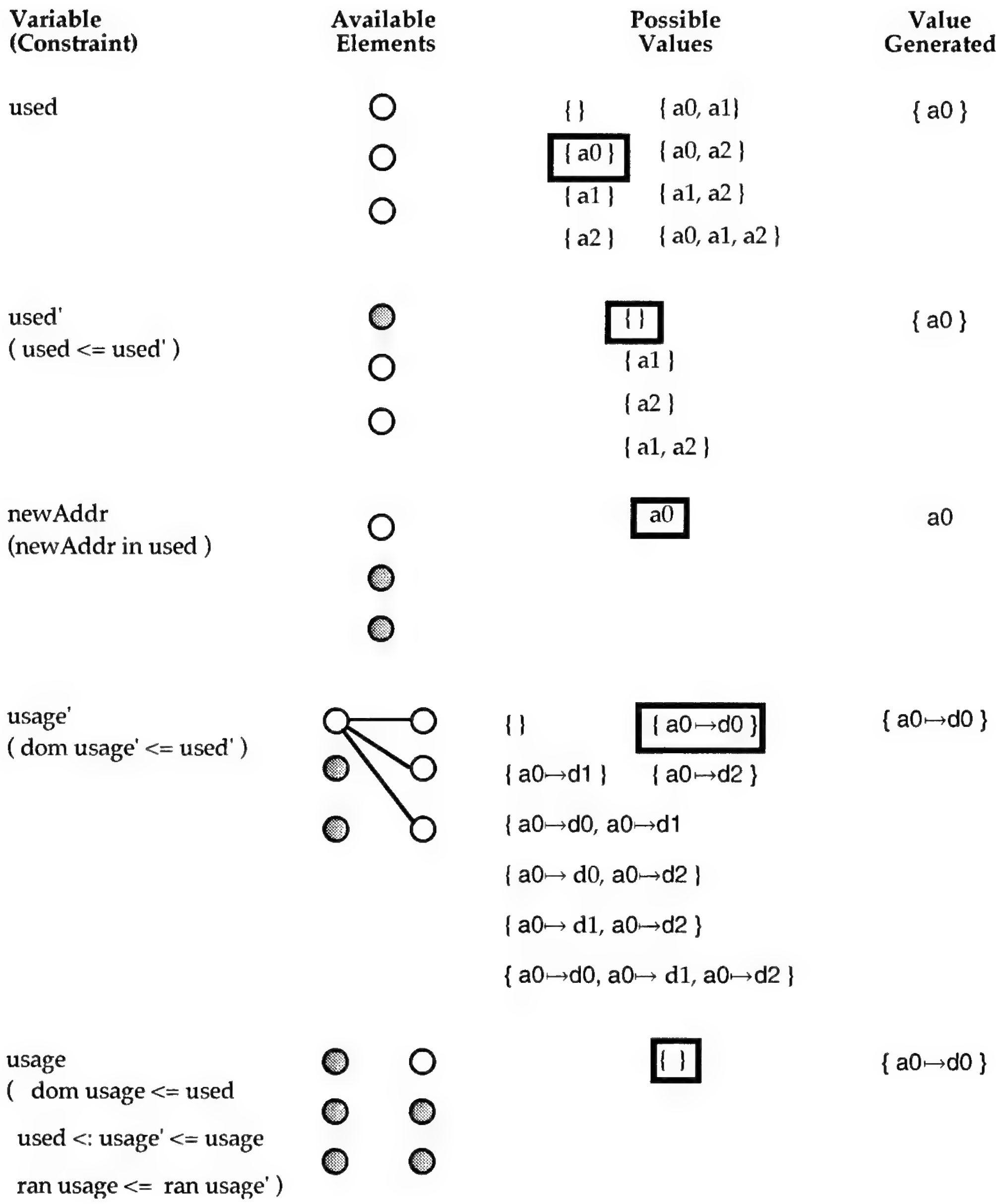


Figure 3.2. An illustration of a single path in a bounded generation search. The triple circles represent the three atomic elements in the universe from which the value may be drawn. Elements represented by circles that have been grayed are not made available in the reduced set of values given to the underlying generator.The boxed value from the reduced list of possible values represents the value chosen by the underlying generator.

The second formula (dom usage' <= used') supports the bounded generation of usage'. The reduced set of values passed to the underlying generator includes only relations that include elements found in used' in their domain.

The first formula (dom usage <= used) has the equivalent effect on the generation of usage as the second formula has on the generation of usage'. Because it constrains usage to be a function, the fourth formula (func usage and (used <: usage') <= usage) supports a further reduction. Any relation including elements from the domain of used <: usage' in its domain is excluded from the reduced set of values. The value of used <: usage' is then unioned into each relation generated by the underlying generator. These two reductions interact very well, with the first reduction removing all elements not in used from the domain and the second removing all elements in the intersection of used and the domain of usage'. The reduced set of values therefore only includes relations whose domain is included in used \ dom usage'. In addition, the fifth formula (ran usage <= ran usage') removes any relation with a range containing elements not found in the range of usage'.

Figure 3.2 illustrates a single path to a solution in this bounded-generation search given a universe of three addresses and three data elements. Each row of the figure is associated with a single variable and shows the process for choosing the value to be bound to that variable in an assignment generated at that level. The first column lists the variable and any filter formulae used by bounded generation. The three (six) circles in the second column represent the domain (domain and range) from which the reduced set of values is constructed. A circle is grayed when the corresponding element is removed from consideration in the reduced set of values. The third column lists the reduced set of values passed to the underlying generator.

As indicated in Figure 3.2, the underlying generator has several possible choices for most variables. I have chosen a specific value (indicated by a heavy box) from the reduced set of values to be bound to an assignment generated by the underlying generator at that level. Before being bound to the assignment yielded by the bounded-generation generator itself, the bounded-generation generator must pass this value to the projection function, which may modify the value. The fourth column indicates the value that will be bound in the assignment yielded by the bounded-generation generator. As an example, the projection function inserts the value of used ({ a0 } in this case) into the value chosen for used'. Bounded generation uses the projection function to similarly modify the value bound to usage before returning that (full) assignment.

3.4 Reduction

This section considers how effectively partial-assignment duplications reduce the size (and cost) of the search for a satisfying assignment. Chapter 2 introduces the concepts of reduction and efficiency as a measure of this effectiveness. For the techniques described in this chapter, the efficiency is always a perfect 1.0 for any filter formula supported by the technique. The reduction, on the other hand, depends both on the problem domain and the filter formula ϕ'. In this section, I develop a heuristic that estimates the reduction for the relational problem domain as a function of the filter formula. Ladybug uses these estimates in choosing a variable ordering.

The reduction of a formula ϕ for a duplication $\approx$ and a set of values υ is given by $R(\approx,\phi,\upsilon)$. The reduction is the ratio of the number of full assignments that contain only values drawn from υ τo the number of equivalence classes in $\approx$ containing at least one assignment drawn entirely from υ. For a partial-assignment duplication, the number of equivalence classes is one more than the number of full assignments that satisfy the filter formula ϕ'.

Computing the exact number of satisfying assignments for a relational formula is in general not possible without the exhaustive search that selective enumeration seeks to avoid. Obtaining reasonable estimates, on the other hand, is a straightforward procedure.

For atomic formulae (or negated atomic formulae), the combination of the grammar production defining the formula and the set of values maps directly to an estimated percentage of satisfying assignments. For example, if the atomic formula involves the scalar equality operator and υ contains k scalars, approximately 1/k of all full assignments will be satisfying assignments. On the other hand, approximately 1/2 of all full assignments will be satisfying assignments if the operator is the set membership operator (in), regardless of the set of values υ.

These approximations assume that the set of values used for any variable includes all well-typed values derived from any atomic elements in the set of values, which I call a *complete* set of values for that variable. A complete set of values passed to a generator for a set variable includes the powerset of the set of all atomic elements in both the set of values itself and the given type of that variable. Similarly, a complete set of values passed to a generator for a relation variable includes all relations that can be built from the cross product of the atomic elements in the set of values and that satisfy the typing requirements of the variable. Ladybug only considers complete sets of values.

Definition 3.6 **(Complete Set Of Values)**

A set of values $\upsilon : \mathbf{P}\,\mathit{Value}$ is a *complete* set of values for a variable v iff

$$((U \cap \upsilon) \cup \mathbf{P}(U \cap \upsilon) \cup \mathbf{P}((U \cap \upsilon) \times (U \cap \upsilon))) \cap \mathit{Typing}(\mathrm{v}) \subseteq \upsilon \subseteq.$$
$$((U \cap \upsilon) \cup \mathbf{P}(U \cap \upsilon) \cup \mathbf{P}((U \cap \upsilon) \times (U \cap \upsilon)))$$

If the set of values is complete and the terms in the formula are simple variables, these approximations will be exact. Assuming the set of values includes the atomic elements e_1, e_2, ..., e_k, the atomic formula a in s will be satisfied by one half of the full assignments, whereas the atomic formula a = b (where a and b are both scalar variables) will be satisfied by 1 / k of the full assignments.

τ	$\#\tau$
v	k/2
$\{ \tau_1 \}$	1
{ }	0
Un	k
τ_1 U τ_2	$(\#\tau_1 + \#\tau_2) - (\#\tau_1 * \#\tau_2)/k$
τ_1 & τ_2	$(\#\tau_1 * \#\tau_2)/k$
$\tau_1 \setminus \tau_2$	$\#\tau_1 - (\#\tau_1 * \#\tau_2)/k$
dom τ_1	$\#d\tau_1$
ran τ_1	$\#r\tau_1$
$\tau_1(\tau_2)$	$\#r\tau_1 * (\#d\tau_1 * \#\tau_2)/k^2$

Table 3.2: Approximating the mean cardinality of set terms. In the approximations, k represents the number of atomic elements in the set of values, $\#\tau_1$ represents the estimated mean cardinality of the set term τ_1 or the estimated mean number of edges in the relation term τ_1, $\#d\tau_1$ represents the estimated mean cardinality of the domain of the relation term τ_1, and $\#r\tau_1$ represents the estimated mean cardinality of the range of the relation term τ_1.

This approach can yield unacceptably inaccurate approximations where the terms are not simple variables. It approximates that the formula a in { b } will satisfied by one half of the full assignments, although this formula obviously is satisfied by the same 1 / k of the assignments that satisfy a = b. For set terms, much of this inaccuracy can be overcome by adjusting for the cardinality of the

τ	$\#d\tau$	$\#r\tau$	$\#\tau$
v	k - 1/2	k - 1/2	$k^2/2$
$\tau_1 \cup \tau_2$	$(\#d\tau_1 + \#d\tau_2)$ - $(\#d\tau_1 * \#d\tau_2)/k$	$(\#r\tau_1 + \#r\tau_2)$ - $(\#r\tau_1 * \#r\tau_2)/k$	$\#\tau_1 + \#\tau_2$ - $(\#\tau_1 * \#\tau_2)/k^2$
$\tau_1 \& \tau_2$	$(\#d\tau_1 * \#d\tau_2)/k$	$(\#r\tau_1 * \#r\tau_2)/k$	$(\#\tau_1 * \#\tau_2)/k^2$
$\tau_1 \setminus \tau_2$	$\#d\tau_1$ - $(\#d\tau_1 * \#d\tau_2)/k$	$\#r\tau_1$ - $(\#r\tau_1 * \#r\tau_2)/k$	$\#\tau_1 - (\#\tau_1 * \#\tau_2)/k^2$
$\tau_1 ; \tau_2$	$\#d\tau_1 * \#d\tau_2/k$	$\#r\tau_2 * \#r\tau_2/k$	$\#\tau_1 *$ $(1 - \#\tau_2/k^2)^{\#\tau_1 / \#d\tau_1} *$ $\#\tau_2/k$
τ_1+	$\#d\tau_1$	$\#r\tau_1$	$(k^2 + \#\tau_1)/2$
$\tau_1\sim$	$\#r\tau_1$	$\#d\tau_1$	$\#\tau_1$
$\tau_1 <: \tau_2$	$(\#\tau_1 * \#d\tau_2)/k$	$(\#\tau_1 * \#r\tau_2)/k$	$(\#\tau_1 * \#\tau_2)/k$

Table 3.3: Approximations for estimating mean cardinalities of relation terms. In the approximations, k represents the number of atomic elements in the set of values, $\#\tau_1$ represents the estimated cardinality of the set term τ_1 or the estimated mean number of edges of the relation term τ_1, $\#d\tau_1$ represents the estimated mean cardinality of the domain of the relation term τ_1, and $\#r\tau_1$ represents the estimated mean cardinality of the range of the relation term τ_1.

terms. The average cardinality of a term can be approximated as a function of the underlying set of values and the term itself. Table 3.2 provides a formula giving an approximation of the mean cardinality for each set term production in the grammar, based on the set of values and an approximation of the average cardinality of each component term. Table 3.3 provides formulae approximating the cardinality of the domain and the range as well as the cardinality of the relation itself for each relation term production.

The approximations of the cardinalities of sets other than the relational image term are exact computations of the mean across all values in a complete set of values. Other computations, including relational image and the relational union, intersection, and difference operators, are exact for some subsets of a complete set of values and remain a reasonable approximation for a complete set of values. Finally, some approximations, such as the average cardinality of the domain and range of relational variables, are simply heuristics that yield reasonable approximations over the size of problems handled by Ladybug.

Table 3.4 indicates how these approximate mean cardinalities are used to predict the portion of assignments that will satisfy various atomic formulae. In general, the filter formula of a partial-assignment duplication may be the conjunction of atomic formulae[4]. For this approximation, I

[4] Because Ladybug normalizes the formula and solves each disjunct separately, the base formula to solve involves only conjunctions. Ladybug will never generate disjunctions in the filter formulae.

ϕ	**Fraction Satisfying**
$\tau_1 = \tau_2$ where $\tau_1, \tau_2 \in \mathit{Term}_{\text{scalar}}$	$1/k$
τ_1 in τ_2 where $\tau_1 \in \mathit{Term}_{\text{scalar}}$ and $\tau_2 \in \mathit{Term}_{\text{set}}$	$\#\tau_2/k$
$\tau_1 = \tau_2$ where $\tau_1, \tau_2 \in \mathit{Term}_{\text{set}}$	$1/2^k$
$\tau_1 <= \tau_2$ where $\tau_1, \tau_2 \in \mathit{Term}_{\text{set}}$	$(\#\tau_2/k)^{\#\tau_1}$
$\tau_1 = \tau_2$ where $\tau_1, \tau_2 \in \mathit{Term}_{\text{rel}}$	$1/2^{kk}$
$\tau_1 <= \tau_2$ where $\tau_1, \tau_2 \in \mathit{Term}_{\text{rel}}$	$(\#\tau_2/k^2)^{\#\tau_1}$
func τ_1	$((k+1)/2^k)^k$

Table 3.4: Approximate fraction of full assignments satisfying atomic formulae. In the approximations, k represents the number of atomic elements in the set of values, $\#\tau_1$ represents the estimated mean cardinality of the set term τ_1 or the relation term τ_1.

assume that the atomic formulae are independent, with the exception of any formulae that is recognized as entailed by another formula. Therefore, the overall fraction of assignments satisfying the combined formulae is assumed to be the product of the fractions determined for each atomic formulae.

Working through a simple example is of value here. I return to the formula used to search for counterexamples to the claim UniqueAddrAlloc from Chapter 1, which appeared originally as (2.2).

```
(2.2)          ( ( ( dom usage = used and dom usage' = used' ) and
                     ( func usage and func usage' ) ) and
                     ( ( ( used <: usage' ) = usage and used' = ( used U { newAddr } ) )
                     and newAddr in used ) )
```

From this formula, Ladybug recognizes eight formulae to be considered:

1) dom usage = used
2) dom usage' = used'
3) func usage
4) func usage'
5) (used <: usage') = usage
6) used' = (used U { newAddr })
7) newAddr in used
8) { newAddr } <= used'

Ladybug also recognizes that the last formula ({ newAddr } <= used') is entailed by the sixth formula (used' = (used U { newAddr })).

Three of these atomic formulae involve equality of sets (dom usage = used, dom usage' = used', and used' = (used U { newAddr })). Assuming that the universe of values includes three scalar elements, the heuristic estimates that one out of every eight (2^3) assignments will satisfy these formulae individually. Similarly, the fifth formula ((used <: usage') = usage) involves equality of relations, so 1/512 ($1/2^{3*3}$) of all assignments are expected to satisfy it. The seventh formula (newAddr in used) uses the in operator and thus the estimated fraction satisfying depends on the estimated cardinality of the set term. The set term (used) is a simple variable with an average cardinality of 3/2. The heuristic gives an estimated satisfying fraction of (3/2)/3 or simply one half. The last formula uses a subset relation, whose satisfying fraction depends on the estimated mean cardinality of both terms. The right hand term (used') is again a simple variable with an estimated mean cardinality of 3/2. The cardinality of the left hand term is exactly 1. Thus the expected fraction of assignments that will satisfy this term is $((3/2)/3)^1$ or simply one half. The heuristic estimates that 1/8 ($((3+1)/(2^3))^3$) of full assignments will satisfy either functional predicate formulae.

When using only short circuiting, Ladybug chooses the ordering for the search as

Ord = < newAddr , used', used, usage, usage'>

Table 3.5 shows the expected and actual results of the search. The first column lists the variables, in the order considered by Ladybug. The second column shows the number of applicable values, assuming a universe of three atomic elements. The third column shows the number of expected assignments computed (before pruning by short circuiting) at each level in the search. The fourth column shows the expected number of assignments generated (after pruning by short circuiting). The final two columns show the actual number of assignments computed and generated by this search when executed by Ladybug with only short circuiting enabled (and without optimizing backtracking — see Chapter 5 for details).[5]

To determine the expected number of assignments computed in the i^{th} row, I multiply the number of assignments generated by the previous row (or one for the first row) by the number of possible values for the variable. The expected number of assignments yielded is the expected number computed multiplied by the product of the fraction of assignments expected to be satisfied by each of the relevant filter formulae. This number is divided by the fraction of assignments that are satisfied by any formulae implied by the set of formulae considered here.

Variable (Type)	# Values	Expected # Assignments Computed	Expected # Assignments Yielded	Actual # Assignments Computed	Actual # Assignments Generated
newAddr (scalar)	3	3	3	3	3
used' (set)	8	24	12	24	12
used (set)	8	96	12	96	12
usage (relation)	512	768	96	768	144
usage' (relation)	512	6,144	12	9,216	144

Table 3.5: Expected and actual results when analyzing the claim UniqueAddrAlloc with a scope of three. These numbers assume that only short circuiting is enabled, without optimized backtracking.

[5] In the searches reported, Ladybug actually used a universe of six elements: three elements of the given type Data and three elements of the given type Addr. As noted earlier, I have chosen to ignore this given type distinction wherever possible. If the two given types were combined, the results in Ladybug for this example would be identical.

For the second row (used'), the number of assignments computed is 3 (the expected number of assignments generated from the previous row) times 8 (the number of values for used'). This result (24) is multiplied by one half (the fraction of assignments expected to satisfy { newAddr } <= used') to obtain the expected number of assignments yielded (12).

For the third row (used), the number of assignments computed is 96 (12 × 8). The expected number of assignments yielded considers three formulae

```
used' = ( used U { newAddr } )
newAddr in used
{ newAddr } <= used'
```

The first two formulae are first fully bound in level 3. The product of the fractions expected to satisfy these formulae ($1/16 = 1/8 \times 1/2$) is multiplied by the number of assignments computed, yielding 6 assignments. However, the formula { newAddr } <= used' has already been factored into the expected number of assignments generated and is implied by used' = used U { newAddr }. Therefore, its expected fraction of satisfying assignments (1/2) is divided into the number of assignments yielded, resulting in the final estimate of 12.

The expected and actual number of assignments are in exact agreement for the first three variables. The inaccuracy for usage arises because the heuristic for satisfaction of set equality does not consider the cardinalities of the sets involved and because the assumption of independence of the formulae is invalid. The formula used = dom usage is the culprit on both counts. The mean cardinalities of each set term is larger than the 3/2 assumed by the fraction satisfying the set equality heuristic, although for differing reasons. The actual mean cardinality of dom usage is approximately 2.6, slightly larger than the 2.5 estimated by the heuristic. The mean cardinality of used in partial assignments being considered at level 4 is 2, significantly different than the predicted 3/2. This larger cardinality is the result of the formula newAddr in used. Because both terms have similarly larger mean cardinalities, they are slightly more likely to be equal than is estimated by the heuristic.

The inaccuracy for the last variable (usage') also arises due to the assumption of independence of the formulae. The formula used' = dom usage' is implied by the conjunction of the other formulae, a relationship missed by the formula discovery process.

Overall, this approach gives reasonable approximations of the actual search performance. Whereas an exhaustive enumeration search requires generating 1,053,192 assignments, including 786,432 full assignments, the heuristic estimated 7,035 assignments would be required including 6,144 full assignments. This estimate matches well to the actual 10,107 assignments generated including 9,216 full assignments, although the number of satisfying assignments discovered is wrong by a factor of 12, a product of the earlier errors.

All the reductions in the previous example were due to the tree-pruning effects of short circuiting. Although the reduction gained by any partial-assignment technique is identical, the heuristic can be improved when derived variables or bounded generation is used. For derived variables, the reduction is simply the number of possible values for the derived variable, as allowed by the set of values and the *Typing* function.

The reduction gained by bounded generation is a factor of the type of variable and the number of values removed in the reduced set of values. Removing a single element from the reduced set of values gives only a slight reduction for a scalar variable, whereas removing all relations containing that element in their domain gives a great reduction. Table 3.6 summarizes the reductions gained for different types of variables by removing r atomic elements from a complete set of values.

Variable Kind	# Original Values	# Reduced Values	Reduction
Scalar	k	k-r	k/(k-r)
Set	2^{k}	2^{k-r}	2^{r}
Relation (range or domain)	2^{k*k}	$2^{k*(k-r)}$	2^{k*r}

Table 3.6: Reductions gained when reducing the set of values by r atomic elements. In this table, k refers to the number of atomic elements in the initial (non-reduced) set of values.

By combining these reductions with the expected mean cardinality computations given earlier, Ladybug is able to estimate the number of assignments that will be generated at each level of the search tree. Table 3.7 gives the expected and actual number of assignments generated using only bounded generation. The first column lists each variable enumerated in the search, in the order enumerated by the search. The second column gives the number of values available for that variable without bounded generation. The third column gives the average amount by which the number of atomic elements in the reduced set of values will be reduced. This column is the product of the estimated cardinality of each set of values excluded. For example, the variable newAddr is reduced by two formulae (newAddr in used' and newAddr in used). Each formula reduces the space to one half of the original (removing 3/2 of the 3 elements, on average). The combination reduces the available space to one quarter the original size or an average reduction of 2.25 elements.

Variable (Type)	# Values	Average # Elements Removed	Average Reduced # Values	Expected # Assignments Generated	Actual # Assignments Generated
used (set)	8	0	8	8	8
used' (set)	8	1.5	2.8	23	27
newAddr (scalar)	3	2.25	0.75	17	27
usage' (relation)	512	1.5 (domain)	23	391	972
usage (relation)	512	2.5 (domain) 1 (range)	2	782	972

Table 3.7: Expected and actual results when analyzing the claim UniqueAddrAlloc with a scope of three. These numbers assume that only bounded generation is enabled.

The fourth column indicates the average number of values expected to remain in the reduced set of values. This fifth column indicates the number of assignments expected to be generated at this level. This number is the product of the average number of values for this variable and the number of assignments expected for the previous level (or 1 for the first row). The final column gives the actual number of assignments generated by the Ladybug search.

As with the short circuiting estimates, the estimates for the performance of bounded generation match well to the actual results achieved. The errors are again traceable to the independence assumption. As an example, the two filter formulae enabling bounded generation for the variable newAddr are not independent. Because used <= used', only newAddr in used actually contributes to the reduction for newAddr. As the effects of the non-independence can increase or decrease the estimated cardinality, such errors tend to offset in practice, yielding reasonable bottom-line estimates.

3.5 Related Work

The bulk of the work done on reducing the cost of searches focuses on what I call partial-assignment reductions. For each of my three partial-assignment techniques, other researchers have developed techniques that gain similar reductions. These other techniques may require a greater expense to achieve this reduction or may interact poorly with other reduction techniques that are part of selective enumeration. Almost all approaches efficiently exploit derived variables, which is the simplest and most obvious technique.

Short circuiting is simply a form of backtracking search, an idea that dates back at least four decades [Wal58]. Van Hentenryck [VHe89] categorizes the approaches used to reduce this search as

- standard backtracking: pruning a subtree when the current partial assignment fails to satisfy a constraint involving only variables already bound[6].
- forward checking: pruning a subtree if, for some unbound variable there does not exist a value that satisfies all constraints involving that unbound variable and a bound variable.
- looking ahead: pruning a subtree if, for some unbound variable there does not exist a value for that variable along with a set of variables binding the other remaining variables that satisfy all constraints involving the unbound variable and one other unbound variable.

Van Hentenryck refers to the latter two cases as *a priori* conditions, because they prune the tree before actually binding the values to variables. Haralick and Elliot [HE80] show that, for a simple model of searching, forward checking is expected to reduce the cost of the search, but looking ahead may actually increase the cost of the search.

If quantifiers are allowed in the formula language, short circuiting incorporates all three approaches; the filter formula gain the effect of forward checking or looking ahead by capturing the unbound variables with existential quantifiers. Ladybug only allows quantifier-free formulae as filter formulae, meaning that short circuiting is limited to standard backtracking.

The *a priori* reductions instead come from bounded generation. Bounded generation exploits a set of constraints that overlaps with those considered by forward checking: bounded generation is limited in the range of formulae that it supports, but can consider formulae that involve multiple bound variables. More importantly, bounded generation performs this pruning without explicit generation of possible values or testing of extra constraints. In addition, forward checking prunes the entire subtree rooted at the current variable, but only if no value can satisfy the next variable. Instead of this all or nothing pruning, bounded generation prunes individual subtrees rooted at the next variable. This pruning prevents the generation of useless values when some, but not all, values can satisfy the next variable. Therefore, for the constraints that support bounded generation, bounded generation is far more efficient and effective. Adding forward checking for the unsupported constraints would probably improve the efficiency of the search, although the published predicted results [HE80;VHe89] indicate that this gain would probably be minimal.

Bounded generation is more similar in its effect to constraint propagation. In classical constraint propagation, all possible values are enumerated for each variable. As variables are bound to specific values, the constraints restrict the values that are permissible for other variables. The values that are not permitted by previous bindings are explicitly removed from the set associated

[6] Van Hentenryck restricts standard backtracking to binary constraints, a common restriction in constraint satisfaction. I have relaxed that requirement for this discussion.

with the variable. This technique works well when the number of values for each variable is relatively small. For the relational formulae solved by Ladybug, the number of possible values can be huge, making an explicit enumeration of the values intractable.

SEM [ZZ96b], Falcon [Zha96], and Finder [Sla94], all model generation tools, handle this difficulty by breaking the large relational variables into many scalar variables, or cells. Each cell represents the result of the function variable for a tuple of input values, like an entry in a standard multiplication table. This approach limits the number of values for any variable to the size of *U*, the universe of elements. This approach can exploit more forms of formulae than can be exploited by bounded generation, including formulae which constrain parts of a relation in terms of other parts of the same relation. This latter opportunity is important to model finding, as the problems may involve only a single n-ary relational variable, making bounded generation useless. This gain comes at the cost of maintaining many sets of values, increasing the complexity of the implementation.

More important, however, is the interaction of this decision with isomorph elimination. With the structure of the relations sacrificed, only a small portion of the isomorphs can be eliminated. Although this would severely limit the effectiveness of this approach for the problems considered by Ladybug, the impact is relatively minor in model generation, as the domain under consideration is often the integers or another relatively rich domain, where the elements may be distinguished and little symmetry is available.

Chapter 4

Isomorph Duplication

This chapter describes how selective enumeration exploits the isomorph duplication. The isomorph duplication is an outgrowth of the observation that any two elements in the same given type are interchangeable. Therefore, permuting elements within an assignment does not change the interpretation of a formula. A permuted assignment duplicates the original assignment.

Much of the work reflected in this chapter is due in large part to previous work, particularly that of Jackson and Jha [JJD96;JJD98]. This chapter recasts that work in terms of selective enumeration, enabling the analysis of the interactions of the isomorph duplication with the other techniques described elsewhere in this dissertation.

The chapter considers two issues largely ignored in the rest of this dissertation: given types and functions. Although not necessary for isomorph elimination, the presence of given types increases the effectiveness of isomorph elimination. Similarly, there is no need to consider functions as distinct from general relations for isomorph elimination. However, the behavior of this commonly used subset of the relations is sufficiently different from other relations under isomorph elimination that separate consideration is warranted.

The first section in this chapter presents an informal, intuitive overview of permutations, demonstrating a path through the search tree for the ongoing Alloc example. The second section formalizes the isomorph duplication. Section 4.3 describes how to compute the reduction gained from the isomorph duplication. The fourth section considers the interaction between the techniques based on the isomorph duplication and those based on the partial assignment duplication. Finally, Section 4.5 describes related work.

4.1 Introduction

This section uses the previously introduced alloc example to develop the intuition behind the isomorph duplication. As a reminder, the alloc specification describes a simple memory allocator in terms of addresses and data bound to those addresses. The example search looks for a counterexample to a claim that a newly allocated address is never already in use. A counterexample to this claim does not depend on which address is bound to which data element. The counterexample does, however, depend on the same address appearing both as previously used (being in the domain of the variable used) and as the newly allocated address (being the value of the variable newAddr).

In demonstrating a single path through this search tree, I consider a universe of atomic values

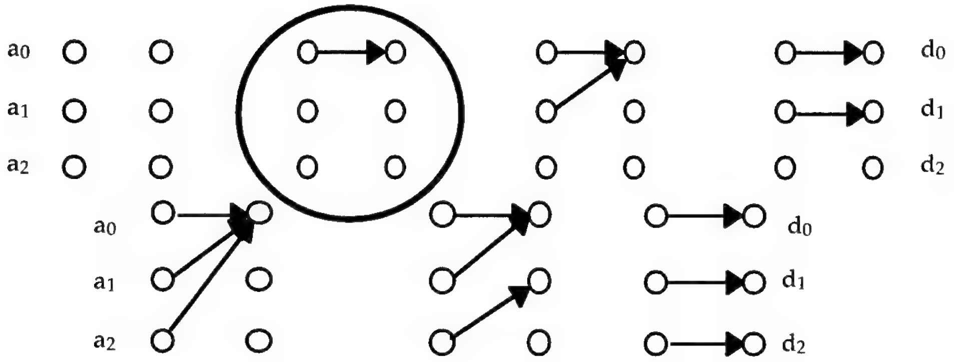


Figure 4.1. Seven functions representing all "interesting" values for the variable usage, with the value chosen for this example circled.

containing three elements of Address (a_0, a_1, and a_2) and three elements of Data (d_0, d_1, and d_2). The search builds a single assignment using the search ordering

Ord = <usage, used, newAddr, usage', used' >

The first step in the search is determining the set of "interesting" values for the first variable, usage. Because usage is a partial function with a domain and range of three elements apiece, there are 64 possible values to consider ($|\text{range}+1|^{|\text{domain}|}$ or 4^3). These 64 values include 9 distinct functions that contain a single edge, mapping only one address to a data element. As noted earlier, which address maps to which data element is insignificant to the counterexample. Therefore, I consider only one of these 9 single-edge functions.

Similarly, I consider only two two-edge functions, one that maps both addresses to the same data element and one that maps the two addresses to two distinct data elements. Three three-edge functions must also be considered: all three addresses mapping to the same data element, two addresses mapping to one data element with the other address mapping to a distinct element, and all three addresses mapping to distinct data elements.

Including the empty function, seven functions represent the entire space of interesting values. Figure 4.1 shows one possible set of interesting values.

To demonstrate a single path through the search tree, I must choose one of these "interesting" values. For this example, I choose the single edge function { $a_0 \mapsto d_0$ }. For a sound search, generating assignments with each of the seven representative functions (or equivalent replacements) is required.

The next step in the search is choosing a value for the variable used. Because used is a set-valued variable with three elements in the domain, there are eight possible values to consider ($2^{|\text{domain}|}$ or 2^3). Without other considerations, four of these sets are interesting: one for each cardinality from zero to three.

However, the function bound to usage earlier in the search differentiates the address a_0 from the other addresses. Therefore two separate single-element sets must be considered: one containing a_0 and one containing an address other than a_0. Similarly, two separate two-element sets must be considered: one containing a_0 (along with one of the other addresses) and one containing two addresses other than a_0. Figure 4.2 shows six sets that represent all the interesting sets for the variable used. Figure 4.2 excludes only two possible values, { a_2 } and { a_0, a_2 }. Because a_1 and a_2 are indistinguishable, these missing values are interchangeable with { a_1 } and { a_0, a_1 }, respectively.

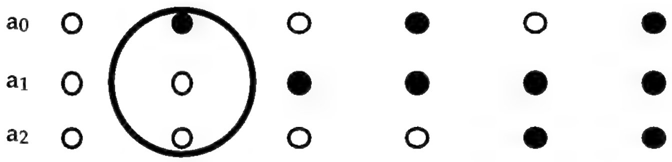


Figure 4.2. Six sets representing all interesting values for the variable used, after considering the effects of the function bound to usage. The set chosen for this example is circled.

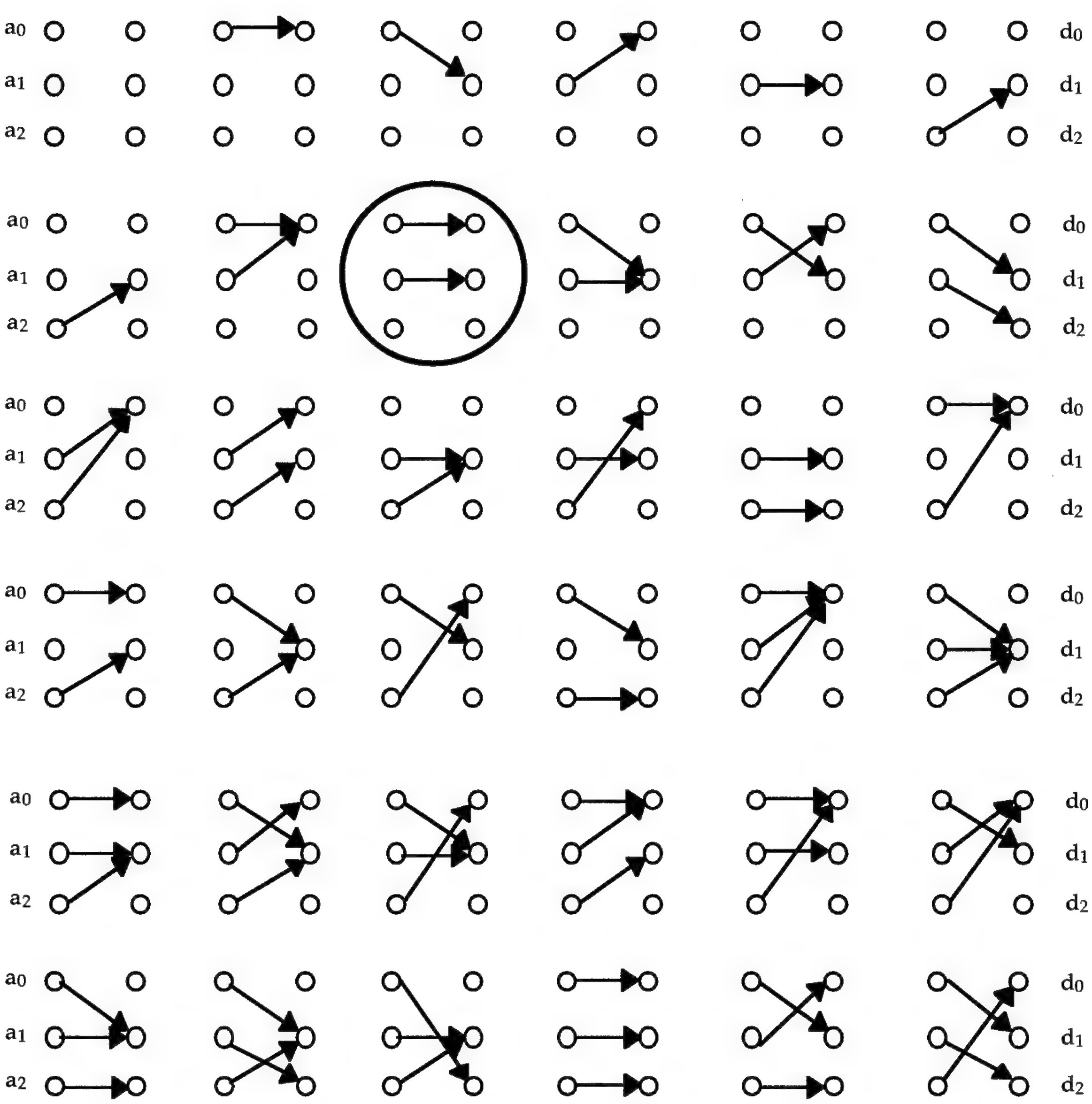


Figure 4.3. The 36 functions representing all interesting values to be considered for the variable usage' for this search. The value chosen is circled.

For this example, I choose the set $\{a_0\}$ to bind to the variable used. Although adding additional variables typically further differentiates the atomic elements, some values yield no further differentiation. The set $\{a_0\}$ differentiates the element a_0 from all other elements of Address, but a_0 has already been differentiated by the value of usage. Therefore, this choice of the value of used does not further differentiate the addresses.

Choosing values for a scalar variable is the simplest of the possible cases. With no differentiation, all elements are indistinguishable and any single element represents all interesting values. If any previous values differentiate the elements, one element from each differentiated set must be considered. When choosing a value for the variable newAddr in this example, two elements must be considered: the differentiated element a_0 and one of the two non-differentiated elements. For this example, I choose the scalar value a_1 as the value of the variable newAddr.

All three addresses are now distinguished. Only two elements of Data, d_1 and d_2, remain indistinguishable. This significantly increases the requirements on the set of functions to be considered for the variable usage'. Without any differentiation, the set of functions to be considered for the variable usage' would consist of the same 7 functions (out of all 64 possible) that I considered for the variable usage. With the differentiation, however, 36 distinct functions must be considered.

Figure 4.3 illustrates a set of functions covering all the interesting functions. As an example, consider the function that binds only a_0 to d_2, which is not included. Because d_1 and d_2 remain undifferentiated, this function is equivalent to the function binding a_0 to d_1, which is included. From this set of interesting functions, I choose the function $\{ a_0 \mapsto d_0, a_1 \mapsto d_1 \}$ to bind to usage' for this example search.

As all elements of Address are now distinguished, all eight possible sets must be considered for the variable used'.

Figure 4.4 illustrates the path through the search considered in this example. From this example, it is obvious that not all values need to be considered for a search to be sound. The reduction in the number of values to be considered depends, in part, on the type of the variable being bound. Functions (and especially relations) can offer greater reductions than sets or scalars. However, this reduction fades as the elements are differentiated by the assignment being constructed.

4.2 Definitions

The reductions illustrated in the previous section are sound because each value that is ignored can be permuted to a value that is already under consideration. This section formalizes this notion of permutation and develops a framework for applying permutations to reduce the search space.

The last section developed the intuitive notion that the elements of a given type can be exchanged without changing the interpretation of the formula. A mapping that exchanges the elements of U is called a *permutation*. Formally, a permutation is any one-to-one total mapping of the atomic elements.

Definition 4.1 **(Permutation)**

A *permutation* $\pi: U \rightarrow U$ is a one-to-one, total mapping of atomic elements.

As a convenience, I denote the special identity permutation that maps each value to itself by parentheses, as in ().

A permutation can be applied to sets and relations by mapping each element within the set or relation with the permutation.

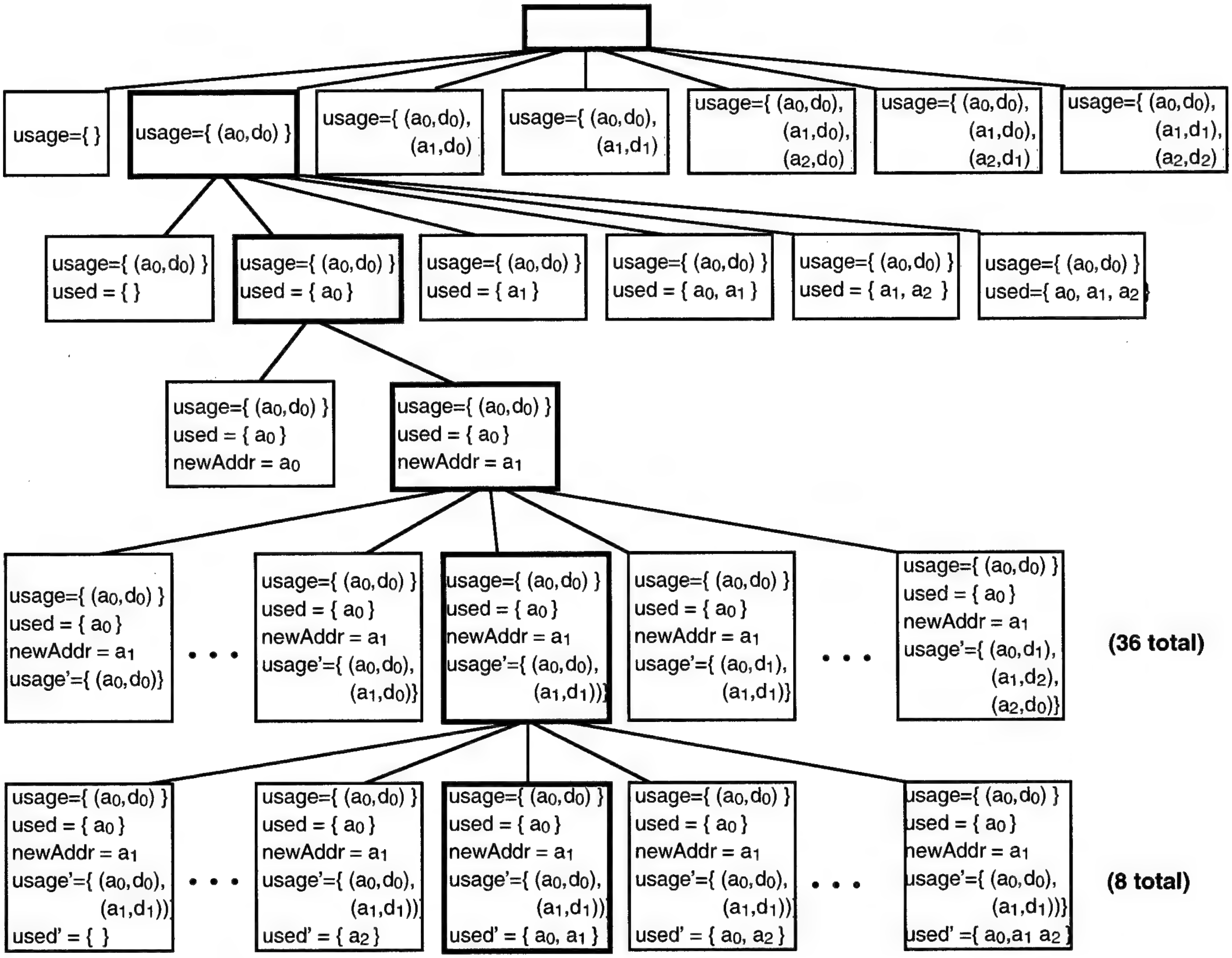

usage={ }
usage={ (a0,d0) }
usage={ (a0,d0), (a1,d0)
usage={ (a0,d0), (a1,d1)
usage={ (a0,d0), (a1,d0), (a2,d0)
usage={ (a0,d0), (a1,d0), (a2,d1)
usage={ (a0,d0), (a1,d1), (a2,d2)
usage={ (a0,d0) } used = { }
usage={ (a0,d0) } used = { a0 }
usage={ (a0,d0) } used = { a1 }
usage={ (a0,d0) } used = { a0, a1 }
usage={ (a0,d0) } used = { a1, a2 }
usage={ (a0,d0) } used={ a0, a1, a2 }
usage={ (a0,d0) } used = { a0 } newAddr = a0
usage={ (a0,d0) } used = { a0 } newAddr = a1
usage={ (a0,d0) } used = { a0 } newAddr = a1 usage'={ (a0,d0)}
• • •
usage={ (a0,d0) } used = { a0 } newAddr = a1 usage'={ (a0,d0), (a1,d0)}
usage={ (a0,d0) } used = { a0 } newAddr = a1 usage'={ (a0,d0), (a1,d1))}
usage={ (a0,d0) } used = { a0 } newAddr = a1 usage'={ (a0,d1), (a1,d1)}
• • •
usage={ (a0,d0) } used = { a0 } newAddr = a1 usage'={ (a0,d1), (a1,d2), (a2,d0)}
(36 total)
usage={ (a0,d0) } used = { a0 } newAddr = a1 usage'={ (a0,d0), (a1,d1))} used' = { }
• • •
usage={ (a0,d0) } used = { a0 } newAddr = a1 usage'={ (a0,d0), (a1,d1))} used' = { a2 }
usage={ (a0,d0) } used = { a0 } newAddr = a1 usage'={ (a0,d0), (a1,d1))} used' = { a0, a1 }
usage={ (a0,d0) } used = { a0 } newAddr = a1 usage'={ (a0,d0), (a1,d1))} used' = { a0, a2 }
• • •
usage={ (a0,d0) } used = { a0 } newAddr = a1 usage'={ (a0,d0), (a1,d1))} used' ={ a0,a1 a2 }
(8 total)

Figure 4.4. The heavy boxes illustrate the path through the search tree described in Section 4.1. At each level the children considered are listed, with the final two levels eliding some of the assignments generated

Definition 4.2 **(Permutations of values)**

A permutation π applied to a value v, given by πv, is defined as

$\pi(v)$ where $v \in \mathit{Value}_{\text{scalar}}$

$\pi(v)$

$\pi(v)$ where $v \in \mathit{Value}_{\text{set}}$

$\{ \pi(x) \mid x \in v \}$

$\pi(v)$ where $v \in \mathit{Value}_{\text{rel}}$

$\{ (\pi(x),\pi(y)) \mid (x,y) \in v \}$

As a convenience, I also allow a permutation to be applied to an assignment. The result of applying a permutation π to an assignment α is an assignment with each variable bound to the corresponding value bound in α, permuted by π.

Definition 4.3 **(Permutations of assignments)**
For any permutation π and assignment α, the value of $\pi\alpha$ is defined as
$\pi\alpha = \{ v \mapsto \pi(\alpha(v)) \mid v \in \text{dom } \alpha \}$

Combining permutations yields new permutations. The resulting permutation has the same effect as applying both the original permutations, in the order specified.

Definition 4.4 **(Product of permutations)**
The product of any two permutations π_1 and π_2, given by $\pi_1\pi_2$, is defined as
$\forall x \in \mathit{Value}.\ \pi_1\pi_2(x) = \pi_2(\pi_1(x))$.

The simplest form of permutation is a *swap*. For any given value, a swap exchanges two atomic elements within that value, leaving all other portions of the value unchanged. A swap is denoted by enclosing the names of the two elements in parentheses. Because the two elements are exchanged, the swaps $(e_0\ e_1)$ and $(e_1\ e_0)$ are equivalent. As a convention, I always list the lexicographically first element first.

Definition 4.5 **(Swap)**
For any two distinct atomic elements e_0 and e_1, the *swap* denoted by $(e_0\ e_1)$ is the permutation that maps e_0 to e_1, e_1 to e_0, and all other elements to themselves.

The compositions of the swaps yield more complicated permutations. Ultimately, any permutation can be constructed from the composition of some set of swaps. The identity permutation, ( ), is represented by the empty sequence of swaps.

Lemma 4.1 For any permutation π, there exists a (possibly empty) sequence of swaps σ_1, σ_2, σ_3, ... σ_k such that $\pi = (\ )\sigma_1\sigma_2\sigma_3 \ldots \sigma_k$.

Proof: Obvious. ■

For convenience, I frequently denote products of swaps using the traditional $(e_0\ e_1 \ldots e_k)$ notation, meaning that e_0 is mapped to e_1, e_1 is mapped to e_2, and so on ending with e_k being mapped to e_0. This notation denotes the permutation defined by the product of swaps $(e_0\ e_1)\ (e_0\ e_2) \ldots (e_0\ e_k)$.

Not all possible permutations are of interest. I define a permutation as *stabilizing* a set of atomic elements if that permutation preserves that set.

Definition 4.6 **(Stabilizing)**
A permutation π *stabilizes* a set s iff $s = \pi s$.

For the remainder of this chapter, I consider only permutations that stabilize the given types. That is, no permutation is considered that swaps elements between two different given types.

A mapping of values that is structure preserving is called an *isomorphism*.

Definition 4.7 **(Isomorphism)**
A one-to-one, total mapping $h{:}\,\mathit{Value} \rightarrow \mathit{Value}$ is an *isomorphism* of *Value* onto itself iff the following conditions hold
$\forall x \in \mathit{Value}_{\text{scalar}}.\ \forall s \in \mathit{Value}_{\text{set}}.\ x \in s \Leftrightarrow h(x) \in h(s)$
$\forall r \in \mathit{Value}_{\text{rel}}.\ \forall t \in \mathit{Value}_{\text{set}}.\ \forall s \in \mathit{Value}_{\text{set}}.\ r(t) = s \Leftrightarrow (h(r))(h(t)) = h(s)$.

That an isomorphism can map only scalars to scalars, sets to sets, and relations to relations follows from this definition. Two values that are mapped by an isomorphism are called *isomorphic* to each other.

Any permutation of the atomic elements is an isomorphism of *Value* when it is applied to values.

Lemma 4.2 Any permutation π is an isomorphism of *Value* onto itself.

Proof: From Definition 4.2. ■

This section began by stating that a search can be sound even if the search ignores assignments that are permutations of ones considered. This is true because permuting an assignment does not affect its truth value for any formulae.

Lemma 4.3 **(Permutation distributes across term interpretation)**
For any term τ, permutation π, set of values υ, and assignment α,
if π stabilizes υ and $Var(\tau) \subseteq \text{dom } \alpha$, then $\pi\, M_{\text{term}}[\tau,\alpha,\upsilon] = M_{\text{term}}[\tau,\pi\alpha,\pi\upsilon]$.

Proof: By structural induction.

For all cases, because π stabilizes υ, $\pi\upsilon = \upsilon$.

If τ is v where $v \in$ *Variable*
Because $Var(\tau) \subseteq \text{dom } \alpha$, $v \in \text{dom } \alpha$
By definition of M_{term}, $v \in \text{dom } \alpha \Rightarrow M_{\text{term}}[\tau,\alpha,\upsilon] = \alpha(v)$.
By definition of $\pi\alpha$, $\pi(\alpha(v)) = (\pi\alpha)(v)$.

if τ is $\tau_1 \cup \tau_2$ where $\tau_1,\tau_2 \in Term_{\text{set}}$
By definition, $M_{\text{term}}[\tau,\alpha,\upsilon] = \{\, x \mid x \in M_{\text{set}}[\tau_1,\alpha,\upsilon] \vee x \in M_{\text{set}}[\tau_2,\alpha,\upsilon] \,\}$.
Because π is an isomorphism, for any set S, $\pi S = \{\, \pi s \mid s \in S \,\}$.
Therefore, $\pi M_{\text{term}}[\tau,\alpha,\upsilon] = \{\, \pi x \mid x \in M_{\text{set}}[\tau_1,\alpha,\upsilon] \vee x \in M_{\text{set}}[\tau_2,\alpha,\upsilon] \,\}$ and
$M_{\text{term}}[\tau,\pi\alpha,\pi\upsilon] = \{\, x \mid x \in M_{\text{set}}[\tau_1,\pi\alpha,\pi\upsilon] \vee x \in M_{\text{set}}[\tau_2,\pi\alpha,\pi\upsilon] \,\}$.
By hypothesis,
$M_{\text{term}}[\tau,\pi\alpha,\pi\upsilon] = \{\, x \mid x \in \pi M_{\text{set}}[\tau_1,\alpha,\upsilon] \vee x \in \pi M_{\text{set}}[\tau_2,\alpha,\upsilon] \,\}$.
Therefore, $\pi\, M_{\text{term}}[\tau,\alpha,\upsilon] = M_{\text{term}}[\tau,\pi\alpha,\pi\upsilon]$.

Other productions follow similarly. ■

Theorem 4.4 **(Permutation preserves formula interpretation)**
For any formula ϕ, permutation π, set of values υ, and assignment α,
if π stabilizes υ and $Var(\phi) \subseteq \text{dom } \alpha$, then $M[\phi,\alpha,\upsilon] = M[\phi,\pi\alpha,\pi\upsilon]$.

Proof: By structural induction.

If ϕ is τ_1 in τ_2 where $\tau_1 \in Term_{\text{scalar}}$ and $\tau_2 \in Term_{\text{set}}$
By definition, $M[\phi,\alpha,\upsilon] = M_{\text{term}}[\tau_1,\alpha,\upsilon] \in M_{\text{term}}[\tau_2,\alpha,\upsilon]$
Because π is an isomorphism,
$M_{\text{term}}[\tau_1,\alpha,\upsilon] \in M_{\text{term}}[\tau_2,\alpha,\upsilon] \Leftrightarrow \pi M_{\text{term}}[\tau_1,\alpha,\upsilon] \in \pi M_{\text{term}}[\tau_2,\alpha,\upsilon]$.
By Lemma 4.3,
$\pi M_{\text{term}}[\tau_1,\alpha,\upsilon] = M_{\text{term}}[\tau_1,\pi\alpha,\pi\upsilon]$ and $\pi M_{\text{term}}[\tau_2,\alpha,\upsilon] = M_{\text{term}}[\tau_2,\pi\alpha,\pi\upsilon]$
By substitution,
$M_{\text{term}}[\tau_1,\alpha,\upsilon] \in M_{\text{term}}[\tau_2,\alpha,\upsilon] \Leftrightarrow M_{\text{term}}[\tau_1,\pi\alpha,\pi\upsilon] \in M_{\text{term}}[\tau_2,\pi\alpha,\pi\upsilon]$.

If ϕ is (ϕ_1 and ϕ_2) where $\phi_1,\phi_2 \in Wff$
By hypothesis.

Other productions follow similarly. ■

If two assignments are guaranteed to yield the same result for a formula, one of the assignments is a duplicate of the other. The *isomorph duplication* considers a pair of assignments to be related if one can be permuted to become the other. Only one assignment from each permutation equivalence class needs to be considered for the search to be sound.

Definition 4.8 **(Isomorph Duplication)**
The isomorph duplication $\approx_\pi$ is defined as $\forall \alpha, \alpha' \in A_N.\ \alpha \approx_\pi \alpha' \Leftrightarrow \exists \pi.\ \pi\alpha = \alpha'$.

The isomorph duplication formally captures the intuition developed in the previous section. That intuition focused on the values considered at each level, whereas the isomorph duplication requires permuting the entire assignment. The intuition captures this requirement by differentiating elements based on the assignment generated thus far. By disallowing swaps involving two elements that have been differentiated, the intuition does not allow any permutations that modify the initial assignment. The set of permutations that leave a single value unchanged is called the *automorphism group* of that value.

Definition 4.9 **(Automorphism Group)**
The automorphism group of a value, given by $Aut(x) : Value \longrightarrow \mathbb{P}Permutation$, is defined as $\forall x \in Value.\ Aut(x) = \{ \pi \mid \pi x = x \}$

If the entire universe of atomic elements is only the four elements of the type $T = \{ t_0, t_1, t_2, t_3 \}$, the automorphism group of the value $\{ t_0, t_1 \}$ includes any combination of the swap of the first two elements and the swap of the last two elements, totaling four permutations:

$$() \qquad (t_0\ t_1) \qquad (t_2\ t_3) \qquad (t_0\ t_1)(t_2\ t_3)$$

Automorphism groups for relations are similar. Consider the universe of values with T as before and $S = \{ s_0, s_1, s_2, s_3 \}$. The automorphism group of the relation $\{ s_0 \mapsto t_0, s_0 \mapsto t_1, s_1 \mapsto t_2 \}$ contains the four permutations

$$() \qquad (s_2\ s_3) \qquad (t_0\ t_1) \qquad (s_2\ s_3)(t_0\ t_1)$$

Because the isomorph duplication considers the effect of permutations on a full assignment, the automorphism group for assignments is required. Each permutation in the automorphism group of an assignment leaves the assignment unchanged.

Definition 4.10 **(Automorphism Group for Assignments)**
The automorphism group of an assignment, given by $Aut(\alpha) : A \longrightarrow \mathbb{P}Permutation$, is defined as $\forall \alpha \in A.\ Aut(\alpha) = \{ \pi \mid \pi\alpha = \alpha \}$

As each permutation in the automorphism group for an assignment must leave each individual value unchanged, the automorphism group of the assignment is exactly the intersection of the automorphism groups of those values.

Lemma 4.5 $\forall \alpha \in A_i,\ Aut(\alpha) = \bigcap_{j=1..i} Aut(\alpha(v_j)) \cap Aut(\emptyset)$

Proof: Obvious. ■

All the pieces are now in place to define isomorph-eliminating generators. As a reminder, a level i generator is a function that takes two arguments: an initial assignment α from A_{i-1} and a universe of values to consider υ. A generator yields a set of assignments from A_i, each of which is identical to the initial assignment α for the first i-1 variables. A sound isomorph-eliminating generator can ignore any assignments that are permutations of an assignment generated. To accomplish this, a level i isomorph-eliminating generator needs to consider a value to be bound to v_i

only if that value is not related to a value bound in another generated assignment by a permutation in the automorphism group of the initial assignment.

Definition 4.11 **(Isomorph-Eliminating Generator)**

A level *i* generator $g(\alpha_0,\upsilon) : A_{i-1} \times \mathrm{P}\,Value \rightarrow \mathrm{P}A_i$ is an *isomorph-eliminating generator* iff

all permutations in $Aut(\alpha_0)$ stabilize υ and

$\forall x \in \upsilon \cap Typing(v_i).\ \exists \alpha \in g(\alpha_0,\upsilon).\ \exists \pi \in Aut(\alpha_0).\ \alpha(v_i) = \pi x.$

An isomorph-eliminating generator is only well-defined if all permutations in the automorphism group of the initial assignment stabilizes the set of values. Otherwise, the generator may generate an assignment where $a(v_i) \notin \upsilon$, violating the requirements of a generator. Fortunately, any permutation that stabilizes the given type sets also stabilizes the standard set of values used by Ladybug.

Computing the exact isomorph-free set of assignments is computationally infeasible for a large universe of elements. The definition of isomorph-eliminating generator allows the generator to generate some isomorphic assignments. This relaxation allows conservative, but efficient, isomorph-eliminating generators to be considered. In the extreme, however, this definition also allows the exhaustive generator to be considered an isomorph-eliminating generator.

Because any permutation in the automorphism group of the initial assignment leaves the initial assignment unchanged, a permutation of each possible level *i* assignment extended from the initial assignment is generated. Therefore, an isomorph-eliminating generator is sound for the isomorph duplication.

Theorem 4.6 An isomorph-eliminating generator $g(\alpha_0,\upsilon)$ is sound for the isomorph duplication $\approx_\pi$ for any formula ϕ if all permutations in $Aut(\alpha_0)$ stabilizes υ.

Proof: Assume α is an element of A_N. $Var_{i-1} \lhd \alpha = \alpha_0 \wedge M[\phi,\alpha,\upsilon] = \mathrm{TRUE}$.

If no such assignment exists, any result of $g(\alpha_0,\upsilon)$ is sound. Otherwise, for g to be sound, there must exist an assignment α' such that $\alpha' \approx_\pi \alpha$ and $Var_i \lhd \alpha' \in g(\alpha_0,\upsilon)$.

By definition, $\alpha' \approx_\pi \alpha$ iff there exists a permutation π such that $\pi\alpha' = \alpha$.

By definition of an isomorph-eliminating generator,

$\exists \alpha'' \in g(\alpha_0,\upsilon).\ \exists \pi \in Aut(\alpha_0).\ \alpha''(v_i) = \pi\alpha(v_i).$

Because $\pi \in Aut(\alpha_0) \wedge Var_{i-1} \lhd \alpha'' = \alpha_0$,

$\alpha'' = Var_i \lhd \pi\alpha.$ ■

Although this section is based solely on permutations, any truth-preserving mapping could be used as the basis for a similar duplication. Therefore, other problem domains may have similar opportunities. However, no other formula-independent truth-preserving mappings are available for the relational problem domain.

Lemma 4.7 For any mapping m such that $M[\phi,\alpha,\upsilon] = M[\phi,m(a),m(\upsilon)]$ for all formulae ϕ, assignments α, and sets of values υ, m is an isomorphism.

Proof: The definition of an isomorphism requires that the formulae

x in s and

r(t) = s

be preserved, where r is a relation variable, s and t are set variables, and x is a scalar variable.

Therefore, any mapping that preserves all possible formulae is an isomorphism. ■

Other mappings may exist, however, that preserve the truth of the formula being solved. For example, for some formulae, it can be shown that any assignment drawn from a sufficiently large universe of atomic elements is equivalent to a "smaller" assignment, drawn on a smaller universe of elements. These "larger" assignments therefore duplicate the "smaller" assignments.

Although Ladybug does not exploit any formula-dependent truth-preserving mappings, these mappings present additional opportunities to trim the search space. Future search engines could exploit this or other formula-dependent mapping duplications.

4.3 Reduction From The Isomorph Duplication

This section focuses on determining the effectiveness of the isomorph duplication. The number of possible permutations is the product of the factorial of the size of each given type. Therefore it is tempting to simply assume that the reduction is the product of these factorials. When examining individual search paths, however, the reduction is somewhat less, sometimes significantly so. To more accurately compute the reduction from the isomorph duplication, I need to introduce some concepts from group theory.

A group is a set of elements with an associative binary operation that is closed over that set of elements. The group must contain an identity element for that operation and each element in the set must have an inverse within the set. An automorphism group is a group, with the elements being permutations and the operation being the product of permutations.

Some subsets of the elements in a group are closed under the operation. If such a subset includes the identity element, it forms a subgroup. According to Lagrange's Theorem, the size of each subgroup is a divisor of the size of the entire group.

A coset is the set of elements obtained by multiplying each element in a subgroup by an element in the full group. A coset is called a left (or right) coset based on the position of the element from the full group in the multiplication. Any two left (or right) cosets of the same subgroup are either identical or disjoint.

Definition 4.12 **(Right coset)**

The right coset G'π where G' is a subgroup of group G and π is an element of G is defined as

$G'\pi = \{ \rho\pi \mid \rho \in G')$

As seen in Section 4.1, each "interesting" value represents some set of possible values, specifically the isomorphism equivalence class containing that value. Therefore, each assignment generated by an isomorph-eliminating generator is equivalent to any assignment in the set of isomorphic assignments that are extensions of the same initial assignment. For a perfect generator, exactly one assignment is generated from each of these sets. The sum of the size of these sets is therefore exactly the size of the set of all possible assignments.

Determining the size of these sets can therefore be used to compute the number of assignments that must be generated by an isomorph-eliminating generator. For any of the "representative" assignments that are actually generated, the set of equivalent assignments can be no larger than the size of the automorphism group of the initial assignment, as this automorphism group includes each permutation considered by the generator. Some of these permutations do not yield distinct assignments. In particular, applying any permutation in the automorphism group of the assignment generated to the assignment yields that assignment.

$()$	(d_0d_1)	(d_0d_2)	(a_0a_1)	(a_0a_2)	$(a_0a_1)(d_0d_1)$	$(a_0a_1)(d_0d_2)$	$(a_0a_2)(d_0d_1)$	$(a_0a_2)(d_0d_2)$
(d_1d_2)	$(d_0d_1d_2)$	$(d_0d_2d_1)$	$(d_1d_2)(a_0a_1)$	$(a_0a_2)(d_1d_2)$	$(a_0a_1)(d_0d_1d_2)$	$(a_0a_1)(d_0d_2d_1)$	$(a_0a_2)(d_0d_1d_2)$	$(a_0a_2)(d_0d_2d_1)$
(a_1a_2)	$(a_1a_2)(d_0d_1)$	$(a_1a_2)(d_0d_2)$	$(a_0a_1a_2)$	$(a_0a_2a_1)$	$(a_0a_1a_2)(d_0d_1)$	$(a_0a_1a_2)(d_0d_2)$	$(a_0a_2a_1)(d_0d_1)$	$(a_0a_2a_1)(d_0d_2)$
$(a_1a_2)(d_1d_2)$	$(a_1a_2)(d_0d_1d_2)$	$(a_1a_2)(d_0d_2d_1)$	$(a_0a_1a_2)(d_0d_1)$	$(a_0a_2a_1)(d_1d_2)$	$(a_0a_1a_2)(d_0d_1d_2)$	$(a_0a_1a_2)(d_0d_2d_1)$	$(a_0a_2a_1)(d_0d_1d_2)$	$(a_0a_2a_1)(d_0d_2d_1)$

Figure 4.5. The permutations in the automorphism group of ∅, arranged with all permutations in each column mapping the relation $\{ a_0 \mapsto d_0 \}$ to the same relation. The universe of elements is assumed to include just the three addresses a_0, a_1, a_2 and the three data elements d_0, d_1, d_2.

Therefore, the number of assignments that are equivalent to an assignment generated is equal to the size of the automorphism group of the initial assignment reduced in some manner by the size of the automorphism group of the assignment generated. To determine the exact form of this reduction for a particular assignment generated, I can construct a table containing the permutations in the automorphism group of the original assignment. The permutations in each column transform the assignment generated into the same assignment.

Figure 4.5 illustrates this table for the first level of the example search illustrated in Figure 4.4 in Section 4.1. The initial assignment is the empty assignment, so the automorphism group of ∅ contains all 36 possible permutations of three addresses and three data elements. The assignment generated in this case is binds the relation $\{ a_0 \mapsto d_0 \}$ to the variable usage.

The first column in this table includes the permutations that transform the assignment into itself, or in other words, the automorphism group of the assignment generated. The second column shows each permutation that maps the assignment generated into a second assignment. In Figure 4.5, these permutations map the assignment generated to the assignment that binds usage to $\{a_0 \mapsto v_1\}$. Intuitively, this column must be the same size as the first column because the two assignments are structurally equivalent and therefore should have the same size automorphism groups. The same argument can be made for each subsequent column, yielding the result that the size of the set of equivalent assignments is exactly the size of the automorphism group of the original assignment divided by the size of the automorphism group of the assignment generated. Each of these columns is a right coset of the automorphism group of the assignment generated.

Lemma 4.8 For any assignment α generated by an isomorph-eliminating generator $g(\alpha_0, \upsilon)$, the size of the set $\{ \pi\alpha \mid \pi \in \mathit{Aut}(\alpha_0) \}$ is $|\mathit{Aut}(\alpha_0)| \,/\, |\mathit{Aut}(\alpha)|$.

Proof: There are two cases to consider: $\mathit{Aut}(\alpha) = \mathit{Aut}(\alpha_0)$ and $\mathit{Aut}(\alpha) \subset \mathit{Aut}(\alpha_0)$.

When the two automorphism groups are identical, the only assignment in the equivalence set is α itself. Therefore, the size of the set is $1 = |\mathit{Aut}(\alpha_0)| \,/\, |\mathit{Aut}(\alpha)|$.

Otherwise, there exists some permutation π in $\mathit{Aut}(\alpha_0)$, but not in $\mathit{Aut}(\alpha)$.

Because $\pi \notin \mathit{Aut}(\alpha)$, $\pi\alpha \neq \alpha$.

Because the identity permutation is in $\mathit{Aut}(\alpha)$, π is an element of the right coset $\mathit{Aut}(\alpha)\pi$.

By definition, each element of the right coset is the product of an element in the subgroup and the chosen element of the full group. More formally,

$$\forall \rho \in \mathit{Aut}(\alpha)\pi.\ \exists \mu \in \mathit{Aut}(\alpha).\ \rho\alpha = (\mu\ \pi)\alpha = \pi(\mu\ \alpha) = \pi\alpha.$$

Therefore, there is a one-to-one correspondence between right cosets of $\mathit{Aut}(\alpha)$ and assignments in the equivalence set.

By Lagrange's theorem, there are $|\mathit{Aut}(\alpha_0)| \,/\, |\mathit{Aut}(\alpha)|$ distinct right cosets of

Aut(α).

Therefore, the size of the equivalence set of assignments is $|\mathit{Aut}(\alpha_0)| \;/\; |\mathit{Aut}(\alpha)|$. ■

Through some algebraic manipulations, the reduction from the isomorph duplication available to a single generator (a single level of the search tree) is the size of the automorphism group of the initial assignment divided by the average of the sizes of the automorphism groups of all assignments that could be generated. As always, the minimum number of assignments generated by a sound generator is the number of possible values divided by the reduction by the duplication for that level of the search tree.

Theorem 4.9 The set of assignments generated by any sound, level *i* isomorph-eliminating generator $g(\alpha_0,\upsilon)$ contains at least

$$\frac{\underset{y \in \upsilon \cap \mathit{Typing}(v_i)}{\mathrm{mean}}(\,|\mathit{Aut}(y) \cap \mathit{Aut}(\alpha_0)|\,) * |\upsilon \cap \mathit{Typing}(v_i)|}{|\mathit{Aut}(\alpha_0)|}$$

assignments.

Proof: Let M be the set of assignments generated by $g(\alpha_0,\upsilon)$.

To be minimal, $\forall \alpha \in M.\ M \cap \{\, \pi\alpha \mid \pi \in \mathit{Aut}(\alpha_0) \,\} = \{\, \alpha \,\}$.

From Lemma 4.8, for any $\alpha \in M$, there are $|\mathit{Aut}(\alpha_0)| \;/\; |\mathit{Aut}(\alpha)|$ assignments in $\{\, \alpha_0 \cup \{v_i \mapsto x\} \mid x \in \upsilon \cap \mathit{Typing}(v_i)\}$.

As these assignments are disjoint for the minimal set, they can be summed, yielding

$$\sum_{\alpha \in M} \frac{|\mathit{Aut}(\alpha_0)|}{|\mathit{Aut}(\alpha)|} = |\, \upsilon \cap \mathit{Typing}(v_i) \,| .$$

Because the size of the automorphism group is identical for each assignment in an equivalence set, the sizes of the automorphism groups can be summed by

$$\sum_{y \in \upsilon \cap \mathit{Typing}(v_i)} |\mathit{Aut}(y) \cap \mathit{Aut}(\alpha_0)| = \sum_{\alpha \in M} \frac{|\mathit{Aut}(\alpha_0)|}{|\mathit{Aut}(\alpha)|} * |\mathit{Aut}(\alpha)| .$$

The two *Aut*(α) on the right hand side cancel out, yielding

$$\sum_{y \in \upsilon \cap \mathit{Typing}(v_i)} |\mathit{Aut}(y) \cap \mathit{Aut}(\alpha_0)| = |\mathit{Aut}(\alpha_0)|\, |M| .$$

Dividing each side by the number of possible values gives

$$\underset{y \in \upsilon \cap \mathit{Typing}(v_i)}{\mathrm{mean}}(\,|\mathit{Aut}(y) \cap \mathit{Aut}(\alpha_0)|\,) = \frac{|\mathit{Aut}(\alpha_0)| * |M|}{|\upsilon \cap \mathit{Typing}(v_i)|} .$$

Isolating for the size of the minimal set of assignments yields

$$|M| = \frac{\underset{y \in \upsilon \cap \mathit{Typing}(v_i)}{\mathrm{mean}}(\,|\mathit{Aut}(y) \cap \mathit{Aut}(\alpha_0)|\,) * |\upsilon \cap \mathit{Typing}(v_i)|}{|\mathit{Aut}(\alpha_0)|}$$ ■

This result gives the reduction for a single-variable search. Computing the reduction for a larger search is somewhat problematic. The exact reduction is a function of the automorphism group of each intermediate assignment generated by the search.

To approximate the reduction for a general search, I first consider the reduction given by a two-variable search. The reduction for this search is given by the reduction for the first level combined with the average reduction given by each invocation of the second level generators. This reduction is given by

$$\frac{|\,Aut(\varnothing)\,|}{\underset{\alpha \in \{v_1 \mapsto x \mid x \in \upsilon \cap \mathit{Typing}(v_1)\}}{\mathrm{mean}}(\,|\,Aut(\alpha)\,|\,)} \quad * \quad \underset{\alpha' \in g(\varnothing,\upsilon)}{\mathrm{mean}}\left(\frac{|\,Aut(\alpha')\,|}{\underset{\alpha'' \in \{\alpha' \cup v_2 \mapsto x \mid x \in \upsilon \cap \mathit{Typing}(v_2)\}}{\mathrm{mean}}(\,|\,Aut(\alpha'')\,|\,)}\right)$$

Although the two averages over the sizes of the automorphism groups of level 1 are not the same, they are close. By replacing the second numerator by the first denominator and cancelling the like terms, I obtain

$$\frac{|\,Aut(\varnothing)\,|}{\underset{\alpha'' \in \{\alpha' \cup v_2 \mapsto x \mid x \in \upsilon \cap \mathit{Typing}(v_2) \wedge \alpha' \in \{v_1 \mapsto x \mid x \in \upsilon \cap \mathit{Typing}(v_1)\}\}}{\mathrm{mean}}(\,|\,Aut(\alpha'')\,|\,)}$$

Generalizing for arbitrary levels, the reduction for the isomorph duplication can be approximated by

$$\frac{|\,Aut(\varnothing)\,|}{\underset{a \in A_N}{\mathrm{mean}}(\,|\,Aut(a)\,|\,)}$$

The average size of the automorphism group for full assignments asymptotically approaches 1 as *N* grows. With only a few variables per given type, the average of the sizes of the automorphism groups of full assignments is already close to one.

The size of the automorphism group of the empty assignment is the product of the factorials of the number of elements in the given type sets. Therefore, as guessed initially, the reduction given by the isomorph duplication can be approximated with

$$R(\approx_{\pi\upsilon},\phi,\upsilon) \cong \prod_{T \in \mathit{Given\ Types}} |\,T\,|\,!$$

Unlike the reduction for the partial assignment duplication, this reduction is nearly independent of the number of variables or the number of known facts about the formula. It is dependent, however, on the number of given types. For this reason, Ladybug redefines the *Typing* relation for the formula to increase the number of given types wherever possible.

Also, unlike the partial assignment duplication, the efficiency of the Ladybug generators for the isomorph duplication is less than the perfect 1.0. Two factors introduce this inefficiency: approximating the automorphism groups and missed permutations in the generators themselves.

4.4 Interactions with Partial Assignment Duplication

This section explores the interaction between the isomorph-eliminating generators and the generators described in Chapter 3 that exploit the partial assignment duplication, focusing on the soundness of a search that utilizes both forms of generator. Chapter 6 discusses the performance implications of these interactions, supported by empirical evidence from the benchmarks.

A search can mix isomorph-eliminating generators with partial-assignment duplication generators in two distinct ways. In the simpler case, the two forms of generator are segregated by variable. In this case, the search for the binding for some variables will exploit a partial-assignment duplication, whereas the search for others will exploit the isomorph duplication, but no search will exploit both. This situation arises with Ladybug when derived variables are enabled as well as isomorph elimination. By Theorem 2.10, the search is sound if each generator is sound for some duplication, even if each generator is sound for a different duplication.

The other form of interaction occurs when an isomorph-eliminating generator is used as the underlying generator for a short-circuiting or bounded-generation generator. The short-circuiting case is again straightforward. Intuitively, short circuiting only removes non-satisfying assignments, so no bad interactions should be expected. More formally, Theorem 3.1 requires only that the underlying generator be sound for some (unspecified) duplication for the short-circuiting generator to be sound for that same duplication. Therefore, as an isomorph-eliminating generator is sound for the isomorph duplication, a short-circuiting generator using an isomorph-eliminating generator as its underlying generator is also sound for the isomorph duplication.

The interesting interaction comes when an isomorph-eliminating generator serves as the underlying generator for bounded generation. As a quick review of bounded generation, bounded generation passes a subset of the set of values to the underlying generator. The underlying generator then produces a representative set of assignments for the reduced set of values. The bounded-generation generator then modifies each assignment by projecting the value bound to the i^{th} variable using a projection function. The rules for defining the reduced set of values and the projection functions are given in Table 3.1.

Bounded generation requires that the underlying generator be limited sound, given a projection function and a reduced set of values. As a reminder, a generator is sound for a duplication if, for any satisfying full assignment, the level *i* prefix of an equivalent full assignment is generated. A generator is limited sound if the projection of a generated assignment is the level *i* prefix of an equivalent full assignment. Definition 3.4 on page 40 formally defines limited soundness.

As I shall demonstrate in this section, any level *i* isomorph-eliminating generator $g(\alpha_0, \upsilon')$ is limited sound with two restrictions on the reduced set of values and the projection function. All the permutations in the automorphism group of the initial assignment must (1) stabilize the reduced set of values and (2) distribute across the proj function. More formally

(1) $\forall \pi \in \mathit{Aut}(\alpha_0).\ \pi$ stabilizes υ'

(2) $\forall \pi \in \mathit{Aut}(\alpha_0).\ \forall x \in \upsilon \cap \mathit{Typing}(v_i).\ \forall x' \in \upsilon' \cap \mathit{Typing}(v_i).$
$\pi(\mathrm{proj}(x', \alpha_0)) = \mathrm{proj}(\pi x', \pi\alpha_0)$

Intuitively, these constraints hold because the reduced set of values υ' and the projection function proj both depend solely on the initial assignment α_0 and each permutation considered leaves α_0 unchanged. Any permutation that leaves α_0 unchanged must also leave the value of any term based solely on variables bound unchanged.

Lemma 4.10 For any assignment $\alpha \in A_i$, term τ, and set of values υ, if $\mathit{Var}(\tau) \subseteq \mathit{Var}_i$ and $\mathit{Aut}(\alpha)$ stabilizes υ then $\mathit{Aut}(\alpha) \subseteq \mathit{Aut}(M_{\mathrm{Term}}[\tau,\alpha,\upsilon])$.

Proof:

By Lemma 4.3, $\forall \pi.\ M_{\mathrm{Term}}[\tau,\pi\alpha,\pi\upsilon] = \pi M_{\mathrm{Term}}[\tau,\alpha,\upsilon]$.

Therefore, $\forall \pi \in \mathit{Aut}(\alpha).\ M_{\mathrm{Term}}[\tau,\alpha,\pi\upsilon] = \pi M_{\mathrm{Term}}[\tau,\alpha,\upsilon]$.

Because π stabilizes υ,
$\forall \pi \in \mathit{Aut}(\alpha).\ M_{\mathrm{Term}}[\tau,\alpha,\upsilon] = \pi M_{\mathrm{Term}}[\tau,\alpha,\upsilon]$.

Therefore, by definition of *Aut*, $\pi \in \mathit{Aut}(M_{\mathrm{Term}}[\tau,\alpha,\upsilon])$. ■

The rules (given in Table 3.1) for producing the reduced set of values under bounded generation fall into two categories: the variable is either (1) a scalar- or set-valued variable or (2) a relation-valued variable. For the scalar/set case, each reduced set is the initial set of values reduced by the value of a term fully bound by the initial assignment. For example, the reduced set of values for the pattern $\tau \Leftarrow v_{\mathrm{set}}$ is $\mathrm{P}M_{\mathrm{set}}[(\mathrm{Un}\backslash\tau),\alpha_0,\upsilon]$. Bounded generation combines reduced sets by intersecting them. The resultant reduced set of values is the set difference between the original set of values and the meaning of the union of terms, each of which is fully bound by the initial assignment. Therefore, Var_{i-1} contains all the variables of this final term.

The relation-valued case is similar, with the reduced set of values being the relations from the original set of values, each domain (or range) restricted by a term fully bound by the initial assignment. The intersection of these reduced sets is described by the set of relations contained in the original set of values, each domain (or range) reduced by the intersection of the domain (or range) reducing term from each reduced set of values. Again, Var_{i-1} contains all the variables of these final reducing terms.

Because the reduced set of values can be expressed as a term made up of the initial set of values, which is stabilized by each permutation in the automorphism group of the initial assignment, and a term fully bound by the first *i-1* variables, each permutation in the automorphism group of the initial assignment stabilizes the reduced set of values used in bounded generation.

Lemma 4.11 For any level *i* bounded-generation generator $g(\alpha_0,\upsilon)$ with a projection function proj and a reduced set of values υ', as described in Section 3.3, each permutation in $Aut(\alpha_0)$ stabilizes υ' if each permutation in $Aut(\alpha_0)$ stabilizes υ.

Proof: To demonstrate $Aut(\alpha_0)$ stabilizes υ', I must demonstrate that

$\forall x \in \upsilon', \forall \pi \in Aut(\alpha_0).\ \pi x \in \upsilon'$,

Each reduced set of values allowed in Section 3.3 includes the elements constructed from one of two forms:

case 1: $M_{\mathrm{Term}}[(\mathsf{Un} \setminus \tau_i),\alpha_0,\upsilon]$

case 2: $M_{\mathrm{Term}}[((\tau_j <: \mathsf{Un}) :> \tau_k),\alpha_0,\upsilon]$

where $Var(\tau_i)$, $Var(\tau_j)$, and $Var(\tau_k)$ are all subsets of Var_{i-1}.

For the first case,

$\upsilon' = \{\, x \setminus M_{\mathrm{Term}}[\tau_i,\alpha_0,\upsilon] \mid x \in \upsilon \cap Value_{\mathrm{set}} \,\}$

Therefore,

$\forall x \in \upsilon \cap Value_{\mathrm{set}}. \exists x' \in \upsilon'.\ x' = x \setminus M_{\mathrm{Term}}[\tau_i,\alpha_0,\upsilon]$.

Because permutation distributes over set difference,

$\forall \pi \in Aut(\alpha_0).\ \forall x \in \upsilon \cap Value_{\mathrm{set}}. \exists x' \in \upsilon'.\ \pi x' = \pi x \setminus \pi M_{\mathrm{Term}}[\tau_i,\alpha_0,\upsilon]$.

By Lemma 4.10, $\forall \pi.\ \pi M_{\mathrm{Term}}[\tau_i,\alpha_0,\upsilon] = M_{\mathrm{Term}}[\tau_i,\pi\alpha_0,\pi\upsilon]$.

Because $Aut(\alpha)$ stabilizes υ, $\forall \pi \in Aut(\alpha_0).\ \pi M_{\mathrm{Term}}[\tau_i,\alpha_0,\upsilon] = M_{\mathrm{Term}}[\tau_i,\alpha_0,\upsilon]$.

Therefore, $\pi x \setminus \pi M_{\mathrm{Term}}[\tau_i,\alpha_0,\upsilon] = \pi x \setminus M_{\mathrm{Term}}[\tau_i,\alpha_0,\upsilon]$.

Therefore,

$\forall \pi \in Aut(\alpha_0).\ \forall x \in \upsilon \cap Value_{\mathrm{set}}. \exists x' \in \upsilon'.\ \pi x' = \pi x \setminus M_{\mathrm{Term}}[\tau_i,\alpha_0,\upsilon]$.

Because $Aut(\alpha_0)$ stabilizes υ,

$\forall \pi \in Aut(\alpha_0).\ \forall x \in \upsilon. \exists x'' \in \upsilon.\ x'' = \pi x$.

Therefore,

$\forall \pi \in Aut(\alpha_0).\ \forall x' \in \upsilon'. \exists x'' \in \upsilon.\ \pi x' = x'' \setminus M_{\mathrm{Term}}[\tau_i,\alpha_0,\upsilon]$.

Therefore, $\pi x' \in \upsilon'$.

The proof for the second case follows similarly. ■

The argument for the constraint on the projection functions follows similarly. Projection functions are also divided into two categories. The simpler projection functions are the identity function, which clearly preserves the permutation. The other projection functions union the value generated by the underlying generator with the value of a term whose variables are contained in Var_{i-1}.

Lemma 4.12 For any level *i* bounded-generation generator $g(\alpha_0,\upsilon)$ with a projection function proj and a reduced set of values υ', as described in Section 3.3,
$\forall x \in \upsilon' \cap \mathit{Typing}(v_i).\ \forall \pi \in \mathit{Aut}(\alpha_0).\ \pi\ \mathrm{proj}(x,\alpha_0) = \mathrm{proj}(\pi x,\alpha_0)$.

Proof: Assume $x \in \upsilon' \cap \mathit{Typing}(v_i)$.

For scalar- and relation-valued variables, $\mathrm{proj}(x,\alpha_0) = x$.

Therefore, $\pi\ \mathrm{proj}(x,\alpha_0) = \pi x$ and $\mathrm{proj}(\pi x,\alpha_0) = \pi x$.

For set- or function-valued variables, proj takes the form
$\mathrm{proj}(x,\alpha_0) = x \cup M_{\mathrm{Term}}[\tau,\alpha_0,\upsilon]$ where $\mathit{Var}(\tau) \subseteq \mathit{Var}_{i-1}$.

Because permutation distributes across union,
$\forall \pi \in \mathit{Aut}(\alpha_0).\ \pi\ \mathrm{proj}(x,\alpha_0) = \pi x \cup \pi M_{\mathrm{Term}}[\tau,\alpha_0,\upsilon]$.

By Lemma 4.10, $\forall \pi \in \mathit{Aut}(\alpha_0).\ \pi x \cup \pi M_{\mathrm{Term}}[\tau,\alpha_0,\upsilon] = \pi x \cup M_{\mathrm{Term}}[\tau,\alpha_0,\upsilon]$.

But $\pi x \cup M_{\mathrm{Term}}[\tau,\alpha_0,\upsilon] = \mathrm{proj}(\pi x,\alpha_0)$. $\blacksquare$

Finally, I prove that these two constraints are sufficient to guarantee that isomorph-eliminating generators are limited sound.

Lemma 4.13 The level *i* isomorph-eliminating generator g for a formula ϕ and a set of values υ is limited sound for the isomorph duplication under a set of values υ' and a projection function proj if

$$\begin{aligned}
&\exists \phi' \in \mathit{Wff}.\ \phi \models \phi' \wedge \mathit{Var}(\phi') \subseteq \mathit{Var}_i \wedge \forall \alpha \in A_N.\ \forall x \in \upsilon \cap \mathit{Typing}(v_i). \forall \pi \in \mathit{Aut}(\mathit{Var}_{i-1} \lhd \alpha).\\
&\qquad \pi \text{ stabilizes } \upsilon' \wedge\\
&\qquad M[\phi',(\mathit{Var}_{i-1} \lhd \alpha) \cup \{ v_i \mapsto x \},\upsilon] = \mathrm{TRUE} \Rightarrow\\
&\qquad\quad \exists x' \in \upsilon'.\ x = \mathrm{proj}(x',\mathit{Var}_{i-1} \lhd \alpha) \wedge \pi x = \mathrm{proj}(\pi x',\alpha)
\end{aligned}$$

where $v_i \in \mathit{Variable}$ and $\mathit{Ord}(v_i) = i$.

Proof: Let $\alpha \in A_N$ such that $M[\phi,\alpha,\upsilon] = \mathrm{TRUE} \wedge \mathrm{ran}\ \alpha \subseteq \upsilon$ and
let $x \in \upsilon \cap \mathit{Typing}(v_i)$ such that $M[\phi',\mathit{Var}_{i-1} \lhd \alpha \cup \{ v_i \mapsto x \},\upsilon] = \mathrm{TRUE}$

Therefore, $\exists x' \in \upsilon'.\ x = \mathrm{proj}(x',\mathit{Var}_{i-1} \lhd \alpha)$.

I need to prove that $\exists \alpha' \in A_N.\ \exists \pi \in \mathit{Aut}(\mathit{Var}_{i-1} \lhd \alpha)$.
$\mathit{Var}_i \lhd \alpha' \in g(\mathit{Var}_{i-1} \lhd \alpha,\upsilon') \wedge \pi(\alpha' \oplus \{ v_i \mapsto \mathrm{proj}(\alpha'(v_i),\mathit{Var}_{i-1} \lhd \alpha)\}) = \alpha$.

If $\mathit{Var}_i \lhd \alpha' \in g(\mathit{Var}_{i-1} \lhd \alpha,\upsilon')$,
then $\mathit{Var}_{i-1} \lhd \alpha' = \mathit{Var}_{i-1} \lhd \alpha$.

Because $\pi \in \mathit{Aut}(\mathit{Var}_{i-1} \lhd \alpha)$, $\pi(\mathit{Var}_{i-1} \lhd \alpha') = \mathit{Var}_{i-1} \lhd \alpha'$.

Therefore, only $\pi(\mathrm{proj}(\alpha'(v_i),\mathit{Var}_{i-1} \lhd \alpha)) = a(v_i)$ is required.

By assumption, $\pi(\mathrm{proj}(\alpha'(v_i),\mathit{Var}_{i-1} \lhd \alpha)) = \mathrm{proj}(\pi\alpha'(v_i),\mathit{Var}_{i-1} \lhd \alpha)$.

Therefore, $\pi\alpha'(v_i) = x'$.

By the definition of an isomorph-eliminating generator,
and because $x' \in \upsilon'$ and $\mathit{Aut}(\mathit{Var}_{i-1} \lhd \alpha)$ stabilizes υ',
$\exists \alpha'' \in A_i.\ \exists \pi \in \mathit{Aut}(\mathit{Var}_{i-1} \lhd \alpha).\ \alpha'' \in g(\mathit{Var}_{i-1} \lhd \alpha,\upsilon') \wedge \alpha''(v_i) = \pi x'$. $\blacksquare$

Finally, tying this all together means that a bounded-generation generator is sound for the isomorph duplication when using any isomorph-eliminating generator as its underlying generator.

Theorem 4.14 A level i bounded-generation generator for a formula ϕ and a set of values υ with an underlying level i isomorph-eliminating generator $g'(\alpha_0,\upsilon')$, set of values υ', projection function proj, and related formula ϕ' is sound for the duplication $\approx_\pi$.

Proof: Obvious from Lemma 4.11, Lemma 4.12, Lemma 4.13, and Theorem 3.4. ■

4.5 Related Work

The idea of removing isomorphs from a search space is very old. Brown et al [BFP88] trace it back to 1874, when Glaisher removed isomorphs in solving the 8-queens problem. Swift [Swi58] coined the term *isomorph rejection* to describe the pruning of isomorphs in several projects that were searching for solutions to interesting mathematical problems or puzzles. He described searches that generated and then removed partial assignments that were isomorphs of other partial assignments generated. This approach of generating and then removing is a common approach in search pruning. Explicit testing by itself can be prohibitively expensive; Freuder [Fre91] describes an algorithm for constraint satisfaction that involves trying all pairs of values for all variables, an algorithm that is exponential in the number of variables. Lam and Thiel [LT89] realized that the cost of generation followed by isomorph testing could exceed the time saved by the reduced number of cases; they suggest only removing isomorphs at selected levels. The approach described in this chapter avoids both generating excess values that will be later removed and any explicit isomorphism tests, at the cost of more expensive generators. These advantages allow the removal of virtually all isomorphs with reasonable cost.

Other researchers have recognized the problem of isomorph rejection. Crawford et al [CG+96] add additional constraints that limit the choices to a single assignment in many equivalence classes. For example, a scalar variable that can vary freely over a set of values can be constrained to a specific value. The challenge in this approach is finding appropriate formulae to constrain a variable, without removing any non-isomorphic cases.

Zhang and Zhang prevent the generation of some isomorphs in SEM [ZZ95b;ZZ96b] with an approach that they call the *Least Number Heuristic*. In SEM, functions are represented as a vector of cells, with each cell representing the value of the function for a particular combination of input arguments. SEM uses these cells as Ladybug uses variables; each layer of the search tree binds possible values to a single cell. Zhang and Zhang note that any value not yet represented in any cell higher in the tree is indistinguishable from any other value not yet represented. Therefore, for the n^{th} cell, they only consider at most the first n values. This approach cannot exploit any symmetries in the structure of the function being generated. While providing a significant reduction at little runtime cost, the least number heuristic leaves a search space with more duplicates than non-duplicates.

Allowing model checkers to exploit the symmetries inherent in the underlying systems is currently a focus of active research [CE+96;ID96;ET99]. Although the specific approaches taken vary between the different systems, all these researchers frame the problem equivalently to the definition of the isomorph duplication in this chapter. None of these approaches uses an explicit search strategy similar to selective enumeration, therefore none considers the level-by-level constructive approach to the permutation group developed in this chapter or considers the problem of generating isomorph-free sets of values.

Other attempts at symmetries have focused on partial-order symmetries [Jha96;BW94], rather the symmetry of values considered here. Partial order symmetries are difficult to discover in relational formulae; the CTL logic used in model checkers and the standard STRIPS framework used in planners both offer more obvious opportunities for partial-order reductions.

Chapter 5

Implementing Ladybug

The previous three chapters describe the theory of selective enumeration and the techniques that allow duplications to be exploited for the relational problem domain. This chapter describes how these techniques have been realized in Ladybug. This chapter focuses on the mechanisms that support the partial-assignment techniques, with only a brief introduction to the mechanisms that support isomorph elimination. I ignore many details of the implementation, focusing only on the algorithms and heuristics that are not obvious in their design or that are surprising in their results.

The first section provides an overview of the Ladybug architecture, giving a context into which the mechanisms can be placed. The second section describes the search function that Ladybug generates to control the search. The third section describes the consequence closure mechanism that Ladybug uses to discover additional candidate filter formulae. The fourth section describes the heuristic used to choose derived variables. The fifth section describes the heuristic used to choose a variable ordering for the search. The sixth section describes Ladybug's generators. The seventh section describes the generation of the search function. The final section describes related work.

5.1 Ladybug Architecture

As illustrated in Figure 5.1, Ladybug consists of two major pieces: the front end and a solver. The front end provides the user interface, the parser, and the thread control. The front end is also responsible for invoking the solver. By default, the front end normalizes the formula into disjunctive normal form. The front end invokes the solver once for each conjunctive formula. For the remainder of this chapter, I assume that the formula being solved is purely conjunctive.

Ladybug is designed to support multiple solvers; the user can choose the solver used in the analysis. In this way, Ladybug can support apples-to-apples comparisons between different approaches to solving relational formulae. For example, a solver could perform a random walk directly over the relational formula produced by the front end, or it could translate that formula into an equivalent boolean formula and employ one of the many existing boolean satisfaction tools. The only solver currently implemented is the selective-enumeration solver.

Along with numerous communication and control methods, each solver provides two major methods: translate and solve. Each solver may choose to do any preparatory work during translate, but only solve is allowed to return any solutions that are discovered. For some solvers, such as the parse tree random-walk approach, translate might perform little actual work. In other solvers, the

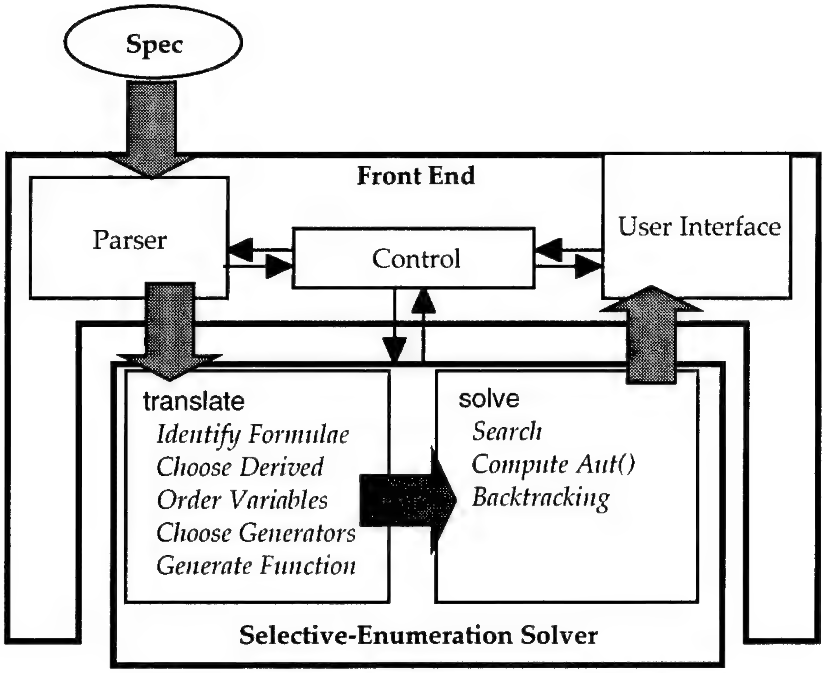


Figure 5.1. The architecture of the Ladybug tool with the selective-enumeration solver "plugged-in".

translate method might embody the majority of the effort undertaken, such as the translation to a boolean formula for a boolean-satisfaction solver.

The selective-enumeration solver makes extensive use of the translate method to improve the chance that the search will quickly locate a counterexample. The result of the translate method is the *search function*, implemented as an easily interpretable structure that embodies all the computations, tests, and appropriate calls to the generators required by a selective-enumeration search. Section 5.2 describes this function and the related virtual machine in more details.

Figure 5.1 illustrates the overall architecture with selective enumeration chosen as the solver. The primary flow of information is:

1) the specification is loaded by the parser,
2) the parser passes parse trees and related structures to the translate method of the solver,
3) the selective-enumeration translate method passes the generated search function to the solve method, and
4) the solve method passes counterexamples to the user interface to display to the user.

The translate method of the selective-enumeration solver works through five steps:

1) identify candidate filter formulae,
2) choose derived variables,
3) choose an ordering for the variables,
4) select and initialize appropriate generators, and
5) generate the search function.

The five sections beginning with Section 5.3 detail these steps.

5.2 Search Function

To understand how these techniques work, it is useful to understand how the generated search function behaves. The search function controls all aspects of the search and performs all computations except the generation of values. In this section, I introduce a textual representation of the search function that Ladybug employs. Figure 5.2 shows the search function used for the alloc example.

A virtual machine, built into Ladybug, executes the instructions that make up the search function. The search function includes three kinds of instructions: computations, tests, and generator invocations. The search function groups these instructions into sections. The search function includes one section for each non-derived variable. Each section begins with the generator invocation for that variable. A successful execution of a generation invocation binds a new value to the associated variable.

As noted earlier in this chapter, the generators implemented in Ladybug behave differently than the idealized generators considered in the previous chapters. Rather than returning a set of assignments, each generator is a Java Enumeration. Each invocation returns a single value.This difference offers two advantages: work is performed only as needed and only a fixed (and small) amount of memory is required for generated values. To maximize the advantage of the fixed memory, every aspect of the search is structured so that no memory allocation occurs during the search.

The computation instructions evaluate all terms considered during the search. Each section includes computations to evaluate each term initially fully bound in that section. One additional section, called the *constant section*, evaluates all terms that are independent of the binding of any values. All computation instructions follow the same format: a typed operator, a target, and optionally one or more operands, as in

TOp *target* [*op1*, *op2*, ...]

In the textual representation, the first letter (or the first two letters) of the opcode indicates the type: "S" for set, "F" for function, "R" for relation, or "G" for given type (scalar). The target and each operand is a memory location; the listing at the top of Figure 5.2 lists the memory locations used other than the variables. A computation instruction implements each derived variable; the target of the instruction is the variable.

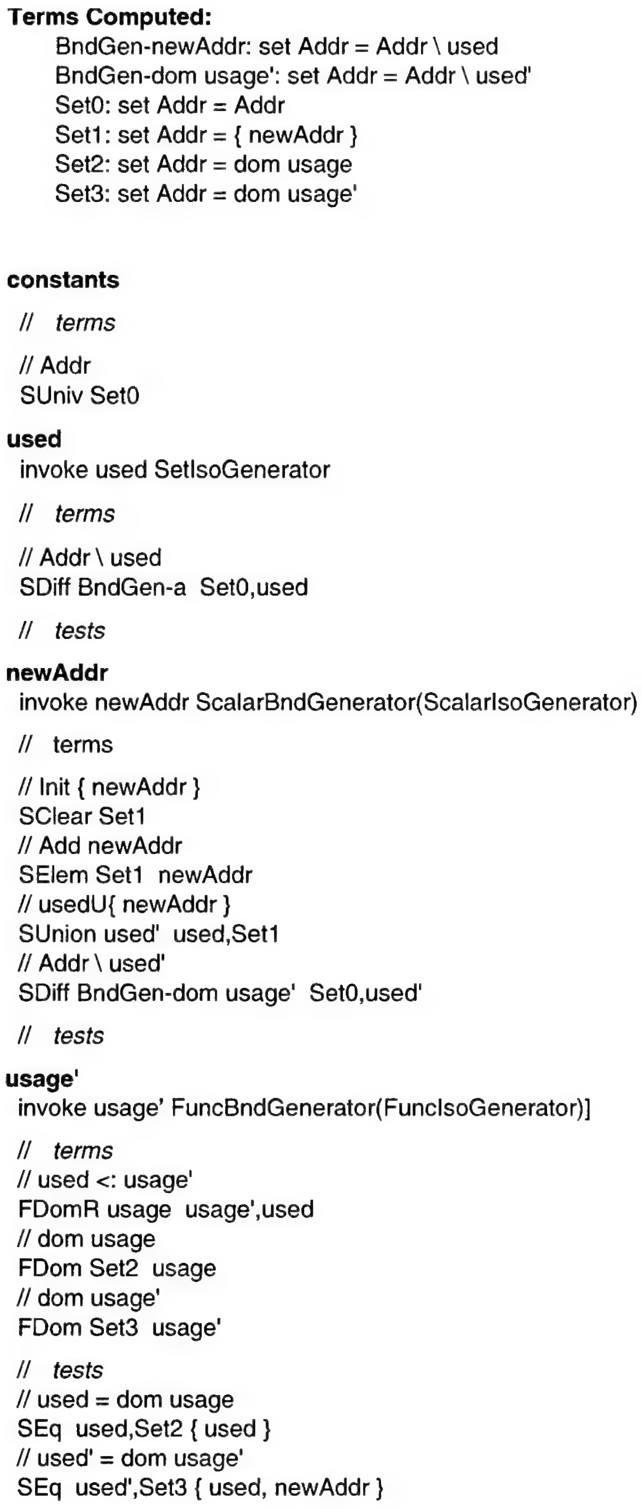
The test instructions evaluate the atomic formulae considered during the search. Each test instruction includes a typed operation, one or more operands, and a set of variables, as in:

TOp *op1* [, *op2*, ...] { [*var1*, *var2*, ...] }

Ladybug uses the set of variables to improve the backtracking, as discussed in Section 5.7. With short circuiting enabled, each section of the search function includes one test instruction for each filter formula used for short circuiting at that level. The remaining constraints are tested at the end of the final section.

During "normal" execution, the virtual machine steps through the instructions in sequential order. However, both the tests and the invocations can branch. A failed test branches back to the invocation that begins the section. If no more values are available for the variable, a generator invocation backtracks to the invocation in a previous section or halts the search.

Figure 5.2 shows the search function for the analysis of the UniqueAddrAlloc claim. The first part



```
Terms Computed:
    BndGen-newAddr: set Addr = Addr \ used
    BndGen-dom usage': set Addr = Addr \ used'
    Set0: set Addr = Addr
    Set1: set Addr = { newAddr }
    Set2: set Addr = dom usage
    Set3: set Addr = dom usage'


constants
 //  terms
 // Addr
 SUniv Set0
used
 invoke used SetIsoGenerator
 //  terms
 // Addr \ used
 SDiff BndGen-a  Set0,used
 //  tests
newAddr
 invoke newAddr ScalarBndGenerator(ScalarIsoGenerator)
 //  terms
 // Init { newAddr }
 SClear Set1
 // Add newAddr
 SElem Set1  newAddr
 // usedU{ newAddr }
 SUnion used'  used,Set1
 // Addr \ used'
 SDiff BndGen-dom usage'  Set0,used'
 //  tests
usage'
 invoke usage' FuncBndGenerator(FuncIsoGenerator)]
 //  terms
 // used <: usage'
 FDomR usage  usage',used
 // dom usage
 FDom Set2  usage
 // dom usage'
 FDom Set3  usage'
 //  tests
 // used = dom usage
 SEq  used,Set2 { used }
 // used' = dom usage'
 SEq  used',Set3 { used, newAddr }
```

Figure 5.2. The search function for the search for a counterexample to the UniqueAddrAlloc claim. Derived variables, bounded generation, isomorph elimination, and optimized backtracking are all enabled.

of Figure 5.2 lists the storage locations used in this search and describes the values that are bound to each location. The first storage location listed, referred to as BndGen-newAddr, is the reduced set of values for the bounded generation of newAddr. The second location listed, referred to as BndGen-dom usage', is the reduced domain for the bounded generation of usage'. The cross product of this set with the set of all elements in Data forms the reduced set of values for usage'. The remaining locations are equivalent to classic compiler temporaries, each holding the result of a simple computation.

The constants section, appearing immediately after the storage section, is the beginning of the code that can be executed by the virtual machine. The code in the constants section computes a single value that is independent of the binding of any variables, the universal set containing all elements in Addr.

The remainder of the search function is separated into sections by the three non-derived variables: used, newAddr, and usage'. The first instruction in each part invokes the indicated generator, storing the returned value, if any, in the indicated variable. The first generator invoked, for the variable used, is the set-valued isomorph-eliminating generator. The remaining two generators are bounded-generation generators based on isomorph-eliminating generators. This relationship is described more thoroughly in Section 5.6.

The third computation instruction in the newAddr section computes the union of the value of used with the value stored in the compiler-generated variable Set1, which is the set containing only the value of newAddr. By storing this result in used', this instruction implements the derived-variable generation of used'.

In this example, the only tests are performed after all the variables are bound, indicating that short circuiting offers no advantages. The first test, checking used = dom usage, does not depend on the binding of the variable newAddr, as indicated by the set of variables shown at the end of the test. The second test, which checks used' = dom usage', does depend on newAddr, as shown in the set of variables. This dependency arises because the value of used' is derived (in part) from the value of newAddr.

5.3 Consequence Closure

All partial-assignment opportunities depend on the knowledge of appropriate filter formulae. Identifying a collection of candidate filter formulae is the first step during translation. This section describes how Ladybug discovers these candidates.

The initial collection of candidate filter formulae is the set of conjuncts that make up the formula being solved. This collection may miss opportunities for short circuiting, derived variables, or bounded generation. To improve the results, Ladybug employs a *consequence closure* mechanism to increase the size of this collection.

Consequence closure recognizes new formulae implied by the current collection. These newly recognized formulae are added to the collection, possibly enabling further recognitions. Ladybug's consequence closure mechanism uses na ad-hoc set of pattern-based rules that generate new formulae when matched. Table 5.1 lists some of the rules used by Ladybug. The complete set of rules appears in Appendix C.

As a simple example, consider a formula that indicates that two sets cover a third set, as in (set1 U set2) = set3. This formula matches the antecedent pattern for the first rule shown in Table 5.1 (**R1**), triggering the generation of the formula (set1 U set2) <= set3.[1] This new formula matches the fifth rule (**R5**), generating two additional formulae: set1 <= set3 and set2 <= set3.

Rule #	Antecedent	Consequent
R1	S0 = S1	S0 <= S1
R2	R0 = R1	R0 <= R1
R3	{ G0 } <= S1	G0 in S1
R4	not G0 in (S1 U S2)	not G0 in S1
R5	(S1 U S2) <= S0	S1 <= S0
R6	not S0 <= (S1 U S2)	not S0 <= S1
R7	R1 <= R0	(dom R1) <= (dom R0)
R8	R1 <= R0	(ran R1) <= (ran R0)
R9	S0 <= (S1 \ S2)	S0 <= (Un \ S2)
R10	S0 <= (S1 & S2)	S0 <= S1
R11	R0 <= (S1 <: R2)	(dom R0) <= S1
R12	R0 <= (S1 <: R2)	R0 <= R2
R13	(S1 U S2) = S0	(S0 \ S1) <= S2

Table 5.1: Selected rules used by the consequence closure mechanism. Any formula discovered that matches an antecedent pattern will generate a formula matching the consequent pattern, with each term in the consequent pattern replaced with the equivalent term from the antecedent pattern. In these rules, any identifier starting with an S represents any term that yields a set, a G any term that yields an element of a given type, and an R any term that yields a relation.

This new formula improves the short circuiting reduction if the variable ordering enumerates set1 and set3 before set2; the new formula set1 <= set3 can short circuit the tree earlier than can the original formula, which requires all three variables to be bound. Alternatively, the new formula set1 <= set3 supports the bounded generation of either set1 or set3, depending on the variable ordering.

Sometimes this mechanism generates unwieldy or seemingly ridiculous formulae. To prevent significant effort being placed on these generated formulae, Ladybug employs an expression simplifier to "clean up" each formula before it is added to the collection. The simplifier replaces terms or entire formulae with simpler terms or formulae that are guaranteed to be equivalent. If a formula simplifies to the constant true, it is ignored. On the other hand, if a formula simplifies to the constant false, the base formula cannot be satisfied and no search is attempted.

The generative rules used by consequence closure are written to guarantee that the mechanism will terminate for any initial collection of formulae. Each term in the antecedent will appear at most once in the consequent, meaning that the generated formula is never longer than the formula triggering the match. Because the number of formulae of any given length is finite, the number of formulae that can be generated is finite.

In practice, the rules typically refer to only two or three terms, so only arrangements of those few terms with the available operators are actually generated. These rules yield a relatively small

[1] The pattern-matching algorithm is aware of the associative and commutative properties of the operators. Therefore, this rule also generates the formula set3 <= (set1 U set2).

Rule #	Antecedent	Consequent
S1	S0 <= S0	**true**
S2	{ } <= S0	**true**
S3	Un <= S0	Un = S0
S4	S0 < S0	**false**
S5	S0 \ { }	S0
S6	S0 U (S1 \ S0)	S0 U S1
S7	S0 \ (S1 \ S0)	S0
S8	dom (S0 <: R1)	S0 & dom R1

Table 5.2: Selected rules used by the expression simplifier.

increase in the number of formulae to consider. Using single antecedent rules, such as those shown, consequence closure typically increases the number of formulae by about a factor of five, with no formula tested showing an increase as large as a factor of nine[2].

Ladybug also supports multiple antecedent rules, which capture concepts such as transitivity. Allowing multiple antecedent patterns in a single rule, however, increases the growth factor significantly (and the cost of discovering them even more significantly). Allowing multiple antecedent rules increases the typical closure time from less than a second to more than an hour. Multiple antecedent rules are therefore impractical, at least in the current implementation, and are disabled by default and for all measurements shown in this dissertation. Unfortunately, multiple antecedent rules are the only rules that can add new opportunities for derived variables.

At this point, an example will help explain the mechanism. This example shows the discovery of candidates from the formula that appeared originally as (2.2) (repeated here for convenience):

```
(2.2)         ( ( ( dom usage = used and dom usage' = used' ) and
                  ( func usage and func usage' ) ) and
                  ( ( ( used <: usage' ) = usage and used' = ( used U {newAddr} ) )
                  and newAddr in used ) )
```

The initial collection of candidate filter formulae is the conjuncts of the formula:

```
dom usage = used
dom usage' = used'
( used <: usage' ) = usage
used' = ( used U {newAddr} )
newAddr in used
func usage
func usage'
```

The first candidate triggers the first rule (**R1**) shown in Table 5.1, yielding the new formulae dom usage <= used and used <= dom usage. Neither of these formulae triggers any further matches. The second candidate formula triggers two equivalent new formulae, dom usage' <= used' and used' <=

[2] Table 5.4 on page 82 lists the results of single antecedent consequence closure on the claims in the benchmark suite.

dom usage'. Again, neither of these new formulae matches any further patterns.

The third candidate formula, (used <: usage') = usage, matches the equivalent relational equality rule (**R2**), yielding two more formulae: (used <: usage') <= usage and usage <= (used <: usage'). These new formulae, in turn, each match rules **R7** and **R8**. These matches directly yield four additional formulae:

```
dom ( used <: usage' ) <= dom usage
ran ( used <: usage' ) <= ran usage
dom usage <= dom ( used <: usage' )
ran usage <= ran ( used <: usage' )
```

Two of these formulae, however, are first simplified by the simplifier using the last rule in Table 5.2 (**S8**). The four formulae added to the candidate collection are

```
used & (dom usage' ) <= dom usage
ran ( used <: usage' ) <= ran usage
dom usage <= used & (dom usage' )
ran usage <= ran ( used <: usage' )
```

The third of these new formulae matches rule **R10** in Table 5.1, yielding two more formulae:

```
dom usage <= used
dom usage <= dom usage'
```

The first of these new formulae is already in the collection, having been introduced as a result of the original formula dom usage = used. The second formula, however, is newly discovered and is added to the collection of candidate filter formulae.

The remaining derivations follow a similar routine. Table 5.3 lists all 27 candidate formulae generated in this process along with a summary of their derivations.

The consequence closure mechanism records the derivation trail for each fact discovered, allowing the search function to test only formulae that are not implied by other formulae already satisfied. The search function shown in Section 5.2 considers only eight of the formulae discovered by this process (1-7 and 10 from Table 5.3) and explicitly tests only two of those formulae (1,2). Ladybug ignores the two functional predicates (6,7) because they are guaranteed by using a generator that yields only functions. By deriving the variables usage and used', Ladybug guarantees the satisfaction of two more formulae (3,4). The final two formulae considered (5,10) support bounded generation, which guarantees their satisfaction in any assignments generated.

For this example, only the trivial weakening of an equality (dom usage' = used' to dom usage' <= used') proved useful. For most of the specifications studied in the benchmark suite, there are more significant effects. Table 5.4 summarizes the effects of consequence closure on the formulae derived from the specifications in the benchmark suite. Chapter 1 provides a brief outline of the benchmarks and Chapter 7 fully describes them.

Table 5.4 gives four pairs of numbers for each claim tested. The first half of each pair represents the number of formulae without consequence closure and the second gives the number with consequence closure. The first pair of numbers indicates the total number of candidate filter formulae discovered, whether actually used or not. The second pair of numbers indicates how many of these formula were chosen to support bounded generation and the third pair indicates how many of these formula were chosen to support short circuiting The fourth pair of numbers gives the total number of filter formula chosen (excluding derived variables), which is always the sum of the corresponding numbers from the second and third pairs. For claims whose original formula included disjunction, the formula was normalized into conjunctive clauses and the numbers describe the last clause.

Formula #	Candidate Formula	Derived From
1	dom usage = used	
2	dom usage' = used'	
3	(used <: usage') = usage	
4	used' = (used U {newAddr})	
5	newAddr in used	
6	func usage	
7	func usage'	
8	dom usage <= used	1(**R1**),16(**R10**)
9	used <= dom usage	1(**R1**)
10	dom usage' <= used'	2(**R1**)
11	used' <= dom usage'	2(**R1**)
12	(used <: usage') <= usage	3(**R2**)
13	usage <= (used <: usage')	3(**R2**)
14	used & (dom usage') <= dom usage	12(**R7,S8**)
15	ran (used <: usage') <= ran usage	12(**R8**)
16	dom usage <= used & (dom usage')	13(**R7,S8**)
17	ran usage <= ran (used <: usage')	13(**R8**)
18	dom usage <= dom usage'	16(**R10**),19(**R7**)
19	usage <= usage'	3(**R12**)
20	ran usage <= ran usage'	19(**R8**)
21	used' <= (used U {newAddr})	4(**R1**)
22	(used U {newAddr}) <= used'	4(**R1**)
23	used <= used'	22(**R5**)
24	{newAddr} <= used'	22(**R5**)
25	newAddr in used'	24(**R3**)
26	used' \ used <= {newAddr}	4(**R13**)
27	used' \ used <= {newAddr}	4(**R13**)

Table 5.3: The candidate formulae generated by the consequence closure mechanism.

As seen by comparing the two numbers in the fourth column, consequence closure improved the number of filter formulae actually chosen for all the claims except those from two specifications: math and phones. The math specification embodies a number of classic mathematical tautologies and, like the phone specification, does not have the rich set of initial candidates typical of a

Specification	Claim	Candidate Filter Formulae	B-Gen Filter Formulae	S. C. Filter Formulae	Total Filter Formulae
alloc	UniqueAddrAlloc	7/27	1/2	2/2	3/4
coda	RCreate	81/322	3/10	40/41	43/51
	RSDRefRen	83/327	0/5	44/50	44/55
digicash	SpendOnce	10/32	1/3	5/6	6/9
faa	X1b_OK	NA	NA	NA	NA
finder	Move	23/111	5/12	9/12	14/24
	TrashingWorks	25/113	6/12	10/14	16/26
HLA owners	AttrDivNot	44/159	2/3	18/19	20/22
	AttrAcqNot	45/170	3/4	19/21	22/25
	CompOwners	134/585	4/13	50/57	54/70
HLA bridge	ObjMapping	30/90	0/2	15/16	15/18
	CheckAcyclicMaps	31/107	0/1	16/18	16/19
math	connex	2/6	0/0	2/2	2/2
	comp	1/1	0/0	1/1	1/1
	shroder	2/8	0/0	1/1	1/1
	closure	1/1	0/0	1/1	1/1
	functions	1/1	0/0	4/4	4/4
mobile IP	loc_update_ok	60/200	2/10	26/28	28/38
phone	CallersCalledP	8/48	0/0	4/4	4/4
styles	FormattingP	34/273	1/14	20/21	21/35

Table 5.4: Summary of the effects of consequence closure on the number of filter formulae. The first number in each column is the number without consequence closure, whereas the second number allows consequence closure. For multiple clause claims, only the final clause is considered.

"real world" specification. Neither of these specifications affords partial-assignment reductions that are comparable in size to those found in the other specifications.

Consequence closure increased the number of filter formulae used by just one in only one claim, UniqueAddrAlloc from alloc. As with the math and phone specifications, alloc has an unrealistically small initial set of candidate formulae.

Consequence closure had a dramatic effect on the final clause of the X1b_OK claim of the faa specification; an inconsistency in the clause was detected and no search was required. Normalization introduced several other clauses of this and other claims with inconsistencies that were detected, but no others were the final clause of a claim.

5.4 Selecting Derived Variables

After the consequence closure mechanism has completed, discovering the list of equalities that support derived variables is straightforward. Any formula that equates a variable to the value of a term not using that variable supports deriving the variable. From the collection of candidate formulae discovered in the previous section, four formulae support derived variables:

```
dom usage = used
dom usage' = used'
( used <: usage' ) = usage
used' = ( used U {newAddr} )
```

Although each of these formulae individually supports derived variables, no single search can use all four formulae to support derived variables. One conflict is immediately obvious: used' can be derived as the value of the term dom usage' or as the value of the term (used U {newAddr}), but not both. Choosing between these two options is straightforward.

The second conflict is less immediately obvious. Deriving the variable used from the value of term dom usage requires that a value is bound to the variable usage before the value of the variable used is computed. Similarly, deriving the variable usage from the value of the term (used <: usage') requires that a value is bound to the variable used before computing the value of usage. Therefore, Ladybug can use only one of these derivations.

Nitpick, the predecessor to Ladybug, simply chose derived variables as enabling formulae were discovered. If Nitpick later discovered a formula that supported deriving a variable but conflicted with a derivation already chosen, Nitpick ignored the opportunity presented by the later formula. This arbitrary selection of derived variables led to instability in the variables that were derived and thus the search time. For at least one specification, mobile IP, a simple rearrangement of the constraints within the specification increased the search time from minutes to hours.

Ladybug, on the other hand, identifies all possible filter formulae for derived variables initially and then attempts to make a good, consistent choice of filter formulae to support derived variables. One constraint and one heuristic guide this choice: no cycles are allowed in the derivation chain and variables whose type includes more values are preferred as derived variables over variables with smaller types. For example, a variable typed as a function with three elements in the domain and three elements in the range has 64 values in its type, whereas a variable typed as a set with three elements in its domain has only 8 values in its type. Therefore, all other things being equal, Ladybug would choose to derive the function rather than the set. Note, however, that deriving one variable may force one or more other variables not to be derived, so the sizes of types of the variables found in the defining terms must also be considered.

To solve these constraints, Ladybug constructs a weighted, directed hyper-graph, with each node representing a variable and each hyper-edge representing a possible derivation. Each edge starts at the variable to be derived by a candidate filter formula and ends at each variable in the term being equated to the derived variable. The weight of an edge approximates the net number of values that can be removed by the derivation. This approximation is computed as the number of values in the type of the variable being derived minus the sum of the number of values in the types of each variable in the defining term. This reduction accounts for the lost opportunity cost of the variables that are not derived.

Figure 5.3 illustrates this hyper-graph for the four candidate filter formulae given earlier. As an example, the edge at the top of the graph represents the derivation used' = (used U {newAddr}).

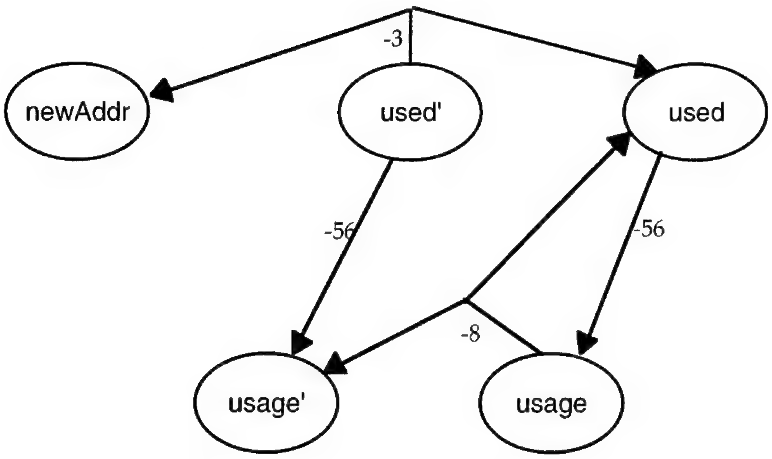


Figure 5.3. The hyper-graph representing the possible derived variables for the search to find counterexamples to the claim UniqueAddrAlloc. This graph includes two edges originating at node representing the variable used' and one each originating from the nodes representing usage and used. Each hyper-edge connects a possibly derived variable to the variables in the term defining the derivation. The weights approximate the savings, with a larger value indicating greater expected savings.

The weight of this edge is the number of values in the type of used' (8) minus the sum of the number of values in the types of used (8) and newAddr (3), for a total of -3.

Any variable that is the starting node for an edge, but is not the terminus for any edge can be derived without placing any restrictions on the other possible derivations. Ladybug begins by choosing the edge with the largest weight originating at one of these "unencumbered" variables. Ladybug uses the formula associated with this edge to support deriving the variable associated with originating node. Ladybug removes the node representing the newly derived variable from the graph, along with any edges that are adjacent to the node.

For the example, used' is the only variable that is a pure source in Figure 5.3. The edge representing used' = (used U {newAddr}), with a weight of -3, has the maximum weight of any edge beginning at used' and is chosen. Therefore, I select the node labeled used' and remove it from the graph, resulting in a new graph shown in Figure 5.4.

When no more variables can be derived using this approach, all possible derivations remaining conflict with another remaining possible derivation. At this point, Ladybug chooses the maximal weight edge remaining in the graph and selects the corresponding formula to support the derivation of the variable associated with the originating node. Ladybug again removes that node from the graph, along with any adjacent edges and the process is continued, checking first for unencumbered variables, and then choosing an edge to break a cycle until all edges have been removed.

The edge with weight -8 is the maximal weight edge in Figure 5.4. Choosing this edge corresponds to deriving the variable usage with the filter formula (used <: usage') = usage. Once usage is removed along with all its adjacent edges, the graph has no more edges and no more variables can be derived.

Once the algorithm terminates with all edges removed, the resulting collection is maximal (no more derivations are possible) and likely to provide a good reduction in the search, although not

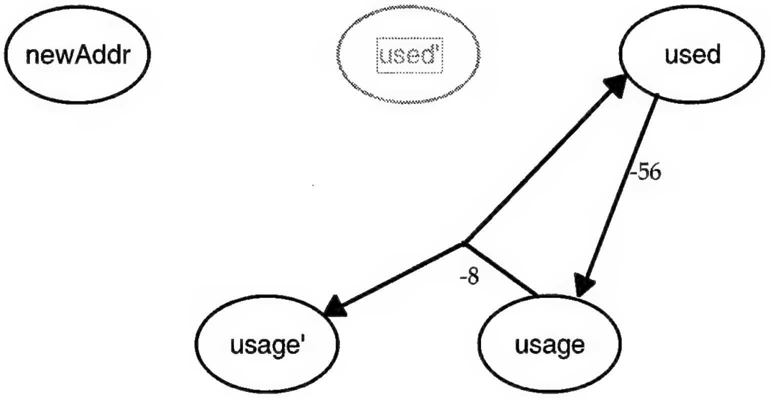


Figure 5.4. The hyper-graph for choosing derived variables after removing the node for used'.

necessarily the largest reduction possible. The first claim is obvious as every possible derivation is represented by an edge and edges are only removed when they are chosen as the basis for a derived variable or they conflict with a derivation chosen. The second claim is based on the intuition that the weighting assigned to the edges is somehow correlated to the reductions actually provided by the derivations. Like greedy algorithms in general, this approach tends to make a good, but not necessarily optimal choice.

5.5 Variable Ordering

Before compiling the discovered formulae into the search function, Ladybug must choose an ordering for the variables. Assuming that the cost of the search is proportional to the number of values generated, one ordering is better than another if it requires fewer values to be generated.

Conceptually, choosing a good ordering is straightforward. For each possible ordering, some of the discovered formulae enable bounded generation, derived variables, or short circuiting. Using the formula-based heuristics developed in Chapter 3 for estimating the partial-assignment reduction, Ladybug can estimate the cumulative effect of each ordering on the size of the reduction offered by the partial-assignment techniques.[3] The ordering offering the largest reduction is chosen as the optimal ordering. Unfortunately, the number of possible orderings increases as the factorial of the number of variables, making this estimation impractical for all but the smallest formulae.

Ladybug uses heuristics to approximate this ordering. Ladybug's first heuristic estimates the savings, in numbers of values not being generated for other variables, of placing a variable at the start of the ordering. Ladybug initially orders the variables in descending order of this estimated savings, refining this ordering with local adjustments.

Ladybug only considers the non-derived variables in this approximation. As implemented in

[3] Although the ordering does not in theory have an effect on the reduction offered by isomorph elimination, there is a practical effect. Ladybug's implementation of isomorph elimination is more efficient for some types of variables than others. In particular, relations with overlapping domains and ranges "lose" many permutations. Generating these values after other variables will improve the efficiency of isomorph elimination. Ladybug currently ignores this consideration.

the search function, each derived variable increases the cost of enumeration for a non-derived variable, but not the number of values generated for that variable. Moving this cost higher in the tree reduces the overall cost of the search, but this difference is insignificant compared to changing the number of values generated. The ordering heuristics therefore consider only the dominant cost of the number of values generated.

The ordering affects the number of values generated by controlling the opportunities for bounded generation and short circuiting. For example, the formula dom usage' <= used' enables the bounded generation of usage' if used' is bound prior to usage' or the bounded generation of used' if usage' is bound prior to used'. The variable used' was already chosen as a derived variable and is therefore not available for bounded generation. The derivation of used' does not depend on usage', so used' may be ordered before usage'. Because used' is a derived variable based on the formula used' = (used U {newAddr}), bounded generation actually requires that usage' must appear after used and newAddr. The heuristics in Chapter 3 estimate that using the bounded generation enabled by the formula dom usage' <= used' will reduce the number of values generated for usage' by approximately a factor of 23.[4] Any variables coming later would also be reduced in number by this same factor.

To approximate the effect of the reduction, the variables required to appear first are given a weight proportional to the effect of the reduction. In this case, the variables used and newAddr accumulate a weight of 23 times the number of possible values for usage'. Therefore, used and newAddr accumulate a weight of 1472 (23*64 possible values for usage'). If the search included other non-derived variables not involved in this reduction, they too would be included in the calculation. The weighting factor is multiplied by the number of possible values for any non-derived variables not involved in the filter formula being considered.

As a second example of the weighting heuristic, consider the formula newAddr in used. If the variable newAddr is ordered before used, bounded generation will reduce the number of values generated for used by a factor of 2. Therefore, Ladybug adds 1024 (2 * 8 values for used * 64 values for usage') to the weighting for newAddr. Similarly, bounded generation will reduce the number of values generated for newAddr by 3/2 if used is ordered before newAddr. Therefore, Ladybug adds a weight of 288 to used (3/2 * 3 values for newAddr * 64 values for usage').

After Ladybug has accumulated the weightings for each variable from each candidate filter formulae, its sorts the variables according to their cumulative weight. In general, an ordering with the higher weight variables preceding the lower weight variables will offer more reductions. For the ongoing example, the weight based ordering is

< newAddr, used, usage' >

However, this ordering is not guaranteed to be optimal. The first variable may improve the reductions of all non-derived variables but the second one and the second variable might offer reduction opportunities for the first variable. If the second variable only contributes to the reduction of the first variable, it will likely be ordered after it, yielding a poor ordering. To improve these local orderings, Ladybug uses a standard bubble sort[5], this time swapping adjacent variables only if swapping them would improve the ordering. The comparison in this sort considers only partial-assignment reductions that depend on a specific ordering of the two variables being com-

[4] Ladybug actually uses a heuristic that takes into consideration that usage' is a function, yielding a smaller reduction estimate of about 6.

[5] Ladybug uses a bubble sort because the bubble sort compares a pair of variables only if they are ordered consecutively. This final sort is attempting to adjust for local inefficiencies in the ordering, so depends on this locality of comparison. The comparison test used in the sort is also meaningful only for these local comparisons.

pared (and is consistent with the remainder of the ordering established, of course).

In the example, when considering the first two variables in the initial ordering, Ladybug looks for candidate filter formulae that involve both used and newAddr, but do not depend on any other variables (as no variables precede them). The only relevant formulae are newAddr in used and {newAddr} <= used. The existing ordering enables a reduction of a factor of 2 in the number of values generated for used. Considered independently, each of these candidate formulae yields a reduction of 3/2 in the number of values generated for newAddr if the order is swapped. Ladybug incorrectly assumes that this leads to a 9/4 reduction in combination and swaps the two variables. In the next step, no advantage is seen when considering swapping newAddr and usage' because no opportunity depends on the ordering of these two variables.

This complete heuristic requires $O(n^2f)$ time to run, where n is the number of non-derived variables and f is the number of candidate formulae discovered by consequence closure. This time is a significant reduction from the $O(n!f)$ cost required for the computing the more obvious ordering.

Although this heuristic chooses a good ordering, the ordering may be flawed for two reasons. First, the algorithm may choose a non-optimal ordering for the estimated reductions. Second, the reductions themselves are estimates. As noted in Chapter 3, where the reduction heuristics were introduced, the heuristics often double count reductions. The double counting occurs more frequently in this algorithm, where every candidate filter formula is considered, often recounting the same basic reductions many times. Despite these problems, the heuristic appears to work well in practice.

5.6 Ladybug Generators

This section describes briefly both the implementation of the generators and how Ladybug chooses which generators will be invoked for each variable. Ladybug considers three factors in this choice: the type of the variable being generated, whether isomorph-elimination has been enabled, and if any filter formulae supporting the bounded generation of the variable have been discovered.

Ladybug includes two sets of generators, each covering a variety of types of values. One set includes isomorph-eliminating generators and the other contains exhaustive generators.[6] The types covered include not only the basic types supported in relational formulae, scalars, sets, and relations, but also subsets of these types, such as functions, injections, and bijections. In choosing the most appropriate generator, Ladybug considers both information supplied in the declarations of the variable and information given by the discovered candidate filter formulae.

The implementation of the exhaustive generators is straightforward, generating all possible combinations. The isomorph-eliminating generators are described later in this section. From the outside, however, the two kinds of generators are indistinguishable, both offering the same interface.

If a candidate filter formula supports bounded generation of the variable, Ladybug initializes a bounded-generation generator. The previously chosen type-based generator becomes the underlying generator. The search function computes the reduced domain and range sets and any projection values and stores them in compiler-defined variables. The bounded-generation generator uses these values to generate each value yielded by the generator. If consequence closure discov-

[6] The exhaustive generators are included both for debugging purposes and to support evaluation of the techniques.

ers multiple formulae that support bounded generation for the variable, the search function computes the intersection of all the reduced domain and range sets and the union of any projection values, allowing one generator to perform the composition of the possible bounded generation.

The isomorph-eliminating generators are among the most complicated portions of the implementation of Ladybug. Implementing isomorph-eliminating generators requires solving two difficult problems: computing the automorphism group and the actual isomorph-eliminating generators. The Ladybug implementation offers a sound approximation to each of these problems: discovering any subset of the automorphism group is sound as is generating additional, isomorphic values.

Ladybug implements two approximations of the automorphism group. Both approximations describe the group as a coloring vector, reflecting the orbits of a subgroup of the full automorphism group. The simpler, and less precise, approximation, which I call *atomic coloring*, considers the subgroup that can be generated from simple swaps. The more precise, but more complicated, approximation called *functional coloring* considers the subgroup generated by any permutation that swaps at most one pair of elements in any given type.

Generating isomorph-free sets of sets or scalars from either form of coloring vector is straightforward. Choosing one element from each coloring class yields a set of isomorph-free scalar values. For a single coloring class, a complete set of isomorph-free sets is obtained by taking the empty set, a set with one element, a set with two elements, and so on up to the set containing the entire coloring class. To generate a set of isomorph-free sets for a complete coloring vector, the cross product of the isomorph-free set of sets from each color is computed.

Isomorph-reduced sets of functions and relations are built from these two simpler isomorph-free generators. For either functions or relations, the first step is to generate the set of isomorphically distinct domains. The isomorph-reducing function generator uses the isomorph-free scalar generator to generate all possible mappings for each element in the domain. The isomorph-reducing relation generator uses the isomorph-free set generator similarly to generate each possible image for each element in each domain. These relation and function generators are not perfect; they do generate some isomorphic values. For the scopes typically considered by Ladybug, however, the number of isomorphs generated is very small, usually less than one per cent of the total relations generated.

5.7 Generating the Search Function

Once the decisions outlined in the previous sections have been made, generating the search function is straightforward. The class representing each operator in the formula language "knows" how to generate the virtual machine code to implement itself.[7] A simple post-order traversal of the parse tree for a term or atomic formula thus yields the complete code required by that expression.

The only question remaining is the order in which to compile these expressions. Other than the obvious variable and inclusion requirements, the possible orderings of the terms are equivalent. Different orderings of the tests, on the other hand, may result in different times required to complete the search. Two factors in the ordering affect the search time: the number of tests evaluated and the target for the next backtrack. By moving the tests most likely to fail earlier in the code, the search time can be reduced slightly.

The second factor, the backtrack target, is far more significant and is the primary selection cri-

[7] To maintain separation between the front end and the selective-enumeration solver, the compilation is actually done by a class hierarchy that is parallel to the parse tree class hierarchy.

teria used by Ladybug in ordering the tests. Ladybug chooses the backtrack target based on a criteria essentially identical to the conflict-directed backjumping algorithm originally developed and described by Prosser in 1993 [Pro93]. In conflict-directed backjumping, when all values for a given variable are exhausted, the backtrack returns to the previous variable that can effect the outcome of one of the tests failed for at least one of the values considered. Obviously, changing the value of any intervening variable will lead to the same tests failing, leading to unproductive subtrees.

Ladybug therefore preferentially orders tests that involve only variables relatively high in the search tree (ignoring the current variable, of course). Ladybug must also consider the variables that are considered by bounded generation; these variables are assumed to also have filtered some values.

The search function tracks the relevant variables with tag sets added to selected instructions in the search function. Ladybug tags the generator invocation with the set of variables involved in bounded generation, if any. Each test has an associated tag set of variables involved in that test. When a generator is reset, the tag set for a variable is reset to the set associated with that invocation. The search function unions the set associated with a test to the variable's set each time a test fails. When a generator exhausts its set of values, the search backtracks to the last variable in the set associated with the current variable.

5.8 Related Work

The consequence closure mechanism uses a simplistic version of unification. Traditional unification systems are more goal directed than the mechanism used here. Adding more goal direction may be one approach to improve the quality and efficiency of the fact discovery.

Many other tools, including the model finding tools [Sla94; ZZ96b], use some form of dynamic fact discovery. Dechter and Frost [DF98] call dynamic fact discovery *learning algorithms*, differentiating the approaches by the amount and quality of information maintained. Slaney maintains extensive information discovered, finding it necessary to reduce the information in two ways to reduce the cost of maintaining the information to a tolerable level. Zhang and Zhang reduce the information maintained significantly to be able to reduce this cost, requiring correspondingly more backtracks in the search as a consequence.

Ladybug depends on static fact discovery to provide information used in choosing a static variable ordering. The static fact discovery and variable ordering allow Ladybug to compile the search function; the compiled search function leads to a significantly faster processing time per assignment.

Other search approaches [SF93;Sla94;ZZ96b;DF98] use heuristics to choose the variable ordering dynamically. The traditional ordering heuristic is to always bind the most constrained remaining variable next. As observed in Section 5.5, the two possible advantages in one ordering over another is 1) the earlier opportunity to backtrack when a variable is constrained to be unsatisfiable and 2) the effect of an early variable on the number of values to consider for the later variables. If all variables still allow some values, dynamic ordering would better reduce the search space by choosing the variable that most constrains other variables. Although this approach would probably choose a similar ordering to the one chosen by Ladybug in many cases, the overall effectiveness of the orderings chosen would presumably be better. In some cases, no static ordering may perform as well as the dynamic ordering chosen. In many cases, the additional information offered by the partial assignment will allow a better choice to be made. These advantages are offset by the additional expense of continually recomputing the ordering.

Another search optimization, value ordering [SF93], is not employed by Ladybug. A value ordering heuristic chooses the next value that is believed to be the most likely to be satisfying or the most constraining for the remainder of the search. Every heuristic attempted to date in Ladybug proved to slow the generation down far more than gain from the search space reduction. An effective and efficient value ordering heuristic is an open research problem.

Chapter 6

Empirical Data

This chapter provides additional empirical data to help evaluate the effectiveness of selective enumeration at reducing the cost of search. The first section describes the benchmark suite of specifications from which the data is drawn. The second section details the behavior of Ladybug when analyzing these specifications with all standard techniques enabled. Section 6.3 considers the behavior of each partial-assignment reduction in isolation and collectively. Section 6.4 considers the behavior of isomorph elimination by itself. Section 6.5 concludes the chapter.

6.1 The Benchmark Suite

I have accumulated a suite of NP specifications to measure the overall effectiveness of selective enumeration for Ladybug as well as the effectiveness of each of the four techniques employed by Ladybug.

Some of these specifications are small, a few with little original purpose other than demonstrating some capability of the analysis. I use these small specifications to demonstrate scaling problems when increasing the size of the scope. Other specifications were developed to analyze real systems and are generally large. These specifications provide a more appropriate measurement of expected real usage and demonstrate issues of scale in number of variables and overall size of specification.

Table 6.1 summarizes the six "real-world" specifications and claims included in the benchmark suite. The first two columns list the specifications and claims checked. The next three columns indicate the number of variables involved in the claim[1], the number of disjunctive clauses in the claim (after normalization), and the average number of atomic formulae per clause. The different clauses for a single formula are much more similar than different and never vary in number of atomic formulae by more than two. Together, these measures give some sense of the relative sizes of the claims. The final column contains a "Yes" if there are any counterexamples to the claim (or executions of the operation). In any realistic usage, many claims checked will be correct; measuring how a tool handles these valid claims is as important as measuring how well it discovers counterexamples. Table 6.2 provides the same information for the five "artificial" specifications.

Appendix B provides a detailed description of each specification, along with the complete text. The following paragraphs summarize the specifications.

[1] The variable count includes both the pre-state and post-state of any variables which might change, as well as any intermediate states expressed in the formulae.

Specification	Claim/Operation	Variables	Clauses	Formulae/ Clause	Satisfiable
coda	RCreate	33	4	59	No
	RSDRefRen	33	6	60	No
digicash	SpendOnce	7	5	4.4	Yes
faa	X1b_OK	10	16	13.75	Yes
HLA owners	AttrDivNot	27	4	28	No
	AttrAcqNot	27	4	29	Yes
	CompOwners	66	1	95	No
HLA bridge	ObjMapping	16	17	17	Yes
	AcyclicObjMaps	1	18	18	No
mobile IP	loc_update_ok	28	1	40	Yes

Table 6.1: Summary of the "real-world" specifications and claims used in the benchmark suite.

Specification	Claim/Operation	Variables	Clauses	Formulae/ Clause	Satisfiable
alloc	UniqueAddr	5	1	5	Yes
finder	Move	14	1	19	Yes
	TrashingWorks	14	1	21	Yes
math	connex	1	2	2	No
	comp	3	1	1	No
	shroder	3	2	2	Yes
	closure	2	1	1	No
	functions	3	1	1	No
phone	CallersCalledP	7	1	7	Yes
styles	FormattingP	14	3	25	Yes

Table 6.2: Summary of the "artificial" specifications and claims used in the benchmark suite.

The coda specification describes the relationship between an idealized model of the volume management for a distributed mobile file system and a model of an actual implementation. This specification is part of a larger one developed by Josh Raiff, a member of the Coda development team. This specification represents the largest overall specification (by number and size of formulae) and the second and third largest claims tested (in terms of state size).

The digicash specification is a simple model of electronic cash, developed by Daniel Jackson. The model is limited to a single bank, a single vendor, and a single customer. The claim tested in the benchmark suite verifies that cash can be spent only once.

The faa specification is another very small specification based on a "real-world" problem. This specification checks part of the proof of the handoff protocol introduced by the United States federal Aviation Agency to pass control of aircraft between flight controllers. This model is limited to a single aircraft and two controllers.

The HLA specifications model aspects of the emerging High Level Architecture standard for distributed simulations, focusing on the ownership management services. Chapter 7 discusses the development and analysis of these specification in greater detail. The HLA owners specification models the ownership managements services as described in an earlier version of the standard and demonstrates why several of the changes were made to the standard. The HLA bridge specification models the effects of the proposed bridge support on the ownership management services. The claims included in the benchmark suite from these two specifications have the largest total search space of any claims tested in the suite.

The final real-world specification is the mobile IP specification. This specification models the mobile IPv6 specification. The claim included in the benchmark suite, loc_update_ok, indicates a flaw in the original specification that allows cycles to be introduced into the forwarding structure.

Three of the remaining specifications have been described previously in this dissertation: alloc, finder, and phone. Another specification, called styles, models the use of inheritance in the style sheet offered by Microsoft Word. All these specifications are smaller than appears to be common for "real-world" specifications, but otherwise appear to reflect similar characteristics to the range of "real-world" specifications. With the smaller size, the suite can consider different scopes for the same claims, an option possible only for one of the "real-world" specifications, digicash.

The remaining specification, called math, is very different from all the other specifications, "real-world" or "artificial". The math specification encodes several known tautologies into NP. This specification offers a difficulty not seen in the other specifications in the suite. The formulae are expressed concisely, meaning that the partial-assignment techniques offer no assistance. This conciseness places the entire burden on isomorph elimination.

6.2 Overall Results

This section presents the results of using Ladybug to analyze the claims and operations indicated in the previous section. For these results in general, the small scope indicates that three elements of each given type were considered, the medium scope indicates four elements, whereas the large scopes indicates five elements. The fourth column in Table 6.3 indicates the exceptions to this general rule. The scope for the faa claim was mandated by the model. The rationale for the scope chosen for the HLA claims is given in Chapter 7.

Specification	Scope	Given Type	# Elements
faa	small	CON	2
HLA owners	small	FED	2
		OBJECT	3
		ATTR	2
		OATTR	6

Table 6.3: Number of elements by given type for selected claims and scope sizes.

Specification	Scope	Given Type	# Elements
		CLASS	1
HLA bridge	small	FED	4
		FEDERATION	2
		OBJECT	3
		BRIDGE	2
		MAP	2
		ATR	1
		OATTR	3
		CLASS	1
	med	FED	7
		FEDERATION	4
		OBJECT	5
		BRIDGE	3
		MAP	4
		ATR	1
		OATTR	5
		CLASS	1

Table 6.3: Number of elements by given type for selected claims and scope sizes.

For each claim or operation analyzed, Table 6.4 lists the number of full assignments in the complete search tree, the number of full assignments checked by Ladybug to cover the complete search space, the number of values in the full search tree, the number of values generated by Ladybug in covering the complete search space, the time required to find the first counterexample (or execution for an operation) if any, and the time to cover the complete search space. All times in this chapter are reported in hours, minutes, and seconds, separated by colons, with the hours or the hours and minutes dropped when unnecessary. For some claims, where the time required to cover the entire search space exceeded 24 hours, no results are given for the complete search space numbers. These claims are AttrAcqNotSoundOwns in hla, the large scope run of SpendOnce in digicash, and the large scope runs of comp, closure, and shroder, all in math.

Three questions can be answered by examining these results:

- How well does selective enumeration work?
- How well does selective enumeration scale?
- When does selective enumeration insufficiently reduce the search space and why?

The remainder of this section focuses on answering these questions.

An underlying question is how to measure and describe the effectiveness. Chapter 2 defines the reduction in the search space in terms of full assignments. The equivalent sense of reduction in terms of values is less elegant, but more indicative of the actual performance of the search. As a

Specification	Claim/ Operation	Scope	# Full Assigns	Assigns Checked	# Values	Values Generated	h:mm:ss to first	h:mm:ss to cover
alloc	UniqueAddr	small	786,432	18	1,053,192	25	0	0
		med	4×10^{8}	39	5×10^{8}	48	0	0
		large	3×10^{11}	77	4×10^{11}	88	0	0
coda	RCreate	small	9×10^{41}	0	1×10^{42}	8,788	—	1
	RSDRefRen	small	9×10^{40}	0	1×10^{41}	17,842	—	1
digicash	SpendOnce	small	1×10^{12}	221,598	1×10^{12}	288,323	0	4
		med	3×10^{20}	2×10^{8}	3×10^{20}	2×10^{8}	10:52	2:22:12
		large	3×10^{30}	—	6×10^{30}	—	1	—
faa	X1b_OK	small*	2×10^{7}	560	2×10^{7}	1,327	0	0
finder	Move	small	4×10^{14}	70	5×10^{14}	215	0	0
		med	7×10^{20}	1,683	9×10^{20}	2,964	0	0
		large	2×10^{27}	37,066	3×10^{27}	52,452	0	0
	TrashingWorks	small	4×10^{14}	30	5×10^{14}	174	—	0
		med	7×10^{20}	658	9×10^{20}	1,824	0	0
		large	2×10^{27}	13,645	3×10^{27}	26,965	0	0.
HLA owners	AttrDivNot	small*	3×10^{50}	0	3×10^{50}	240,614	—	3
	AttrAcqNot	small*	3×10^{50}	—	3×10^{50}	—	2	—
	CompOwners	small*	3×10^{123}	0	3×10^{123}	252,330	—	3
HLA bridge	ObjMapping	small*	2×10^{18}	20,595	2×10^{18}	42,937	0.0	1.0
	AcylicObjMaps	small*	2×10^{18}	20,389	2×10^{18}	42,643	—	1.0
	AcylicObjMaps	med*	5×10^{40}	2×10^{9}	5×10^{40}	2×10^{9}	—	21:15:08
math	connex	small	1,024	372	1,024	372	—	0
		med	131,072	25,130	131,072	25,130	—	0
		large	7×10^{7}	5,538,962	7×10^{7}	5,538,962	—	29
	comp	small	1×10^{8}	281,264	1×10^{8}	284,467	—	4
		med	3×10^{14}	3×10^{9}	3×10^{14}	3×10^{9}	—	8:02:19
		large	4×10^{22}	—	4×10^{22}	—	—	—
	closure	small	262,144	91,300	262,656	91,486	—	1
		med	4×10^{9}	8×10^{8}	4×10^{9}	8×10^{8}	—	4:27:03
		large	1×10^{15}	—	1×10^{15}	—	—	—
	shroder	small	3×10^{8}	254,616	3×10^{8}	257,766	0	6
		med	6×10^{14}	5×10^{9}	6×10^{14}	5×10^{9}	0	21:31:52
		large	8×10^{22}	—	8×10^{22}	—	0	—
	functions	small	262,144	2,189	266,304	2,285	—	1
		med	2×10^{8}	58,192	2×10^{8}	58,667	—	1
		large	5×10^{11}	1,843,429	5×10^{11}	1,845,962	—	54
mobile IP	loc_update_ok	small	2×10^{37}	605,901	3×10^{37}	2×10^{8}	2	1:34:12
phone	CallersCalledP	small	4×10^{13}	804	6×10^{13}	974	0	0
		med	2×10^{23}	65,830	2×10^{23}	68,648	0	1
		large	2×10^{35}	1×10^{7}	3×10^{35}	1×10^{7}	0	2:19
styles	FormattingP	small	4×10^{18}	57	6×10^{18}	2,610	0	0
		med	1×10^{28}	1,160	1×10^{28}	63,686	2	4
		large	2×10^{38}	24,837	2×10^{38}	1,850,891	1:00	1:49

Table 6.4: Results of Ladybug checking the claims in the benchmark suite with all techniques enabled.

simple example of this improvement, consider a search over many variables that short circuits every path at the next to last variable. No full assignments will be generated, but a huge amount of effort may be expended in the search.

I use the value reduction (the total number of nodes in the search tree divided by the number of values generated) as the basic metric for evaluating the techniques. However, this number is only meaningful in the context of the search; a reduction of a billion would be very good if the search tree contains only a few billion nodes, but would be practically useless for a search tree containing 10^{100} nodes. To account for the context and the exponential nature of the problem, I use the ratio of the logarithm of the reduction to the logarithm of the total number of values in the complete search tree. A ratio of 1.0 indicates that the search space is reduced to a single value, the limit on search effectiveness. A ratio of 0.0 indicates that no reduction occurred.

As an example, Ladybug must generate 8,788 values of the 10^{42} values in the complete search tree for the RCreate claim in the Coda specification. Selective enumeration therefore reduces this search by a factor of $1.1\text{x}10^{38}$ (10^{42} divided by 8,788). The log (base ten) of this reduction is 38.1.The log of number of values is 42.0, yielding a ratio of 0.91 (38.1/42.0). If this ratio is relatively constant across different scopes, the actual reduction is growing exponentially with the size of the problem. The problem is NP-complete, so no polynomial solution is expected. A constant ratio indicates a constant reduction in the exponent; for the case of RCreate, the exponent is approximately divided by 11. This reduction does not remove the ultimate cliff beyond which analysis is intractable, but it does shift the cliff to a larger scope. The divisor determines the size of the total search space that is tractable. Depending on the user's patience, enumerating about one hundred million values is the limit of practical usage for Ladybug. For the ratio for the RCreate claim, this corresponds to a tractable total search tree containing about 10^{90} values ($10^{90/11} = 10^{8.2}$).

Returning to the first question, selective enumeration clearly works well for almost every test in the benchmark suite. Within seconds, Ladybug either returns a counterexample or validates the claim for the initial scope for every test. For a few tests, especially the claims from the math specification, the larger scopes pushes Ladybug into minutes or even days. The third question in this section focuses on those specifications where the analysis becomes intractable.

The ratio described above provides a more concrete measure of effectiveness than any simple success or failure in a fixed period of time. Figure 6.1 shows the reduction ratios for each benchmark test. The total length of each line represents the ratio for Ladybug when checking the full state space. Most of the bars consist of two bars, one solid and one more slender that is gray or filled with slashes. The solid bar is the reduction ratio for Ladybug with only the partial-assignment techniques enabled. The slashed bar is the reduction ratio for Ladybug with only isomorph-elimination enabled, shifted to the right of the full reduction bar. Some of these slashed bars are estimates, as the full search with only isomorph elimination is infeasible.[2] If neither the isomorph elimination or the partial-assignment techniques reduces the search to feasible levels, a single unfilled bar represents the full ratio only. The number at the right of each bar is the reduction ratio achieved by the combination of all techniques.

Excluding the math claims, which have no reduction from partial-assignment techniques, Ladybug has a reduction ratio between 0.75 and 0.90 for most of the tests. This ratio means that Ladybug usually reduces the exponent of the number of values generated by a factor between four and ten. Reducing the exponent allows Ladybug to successfully analyze most searches of state spaces of 10^{32} to 10^{80}, including most of the benchmark suite.

[2] I confirmed the accuracy of the estimates against the feasible searches. In every case, the estimate was within .01 of the reduction ratio measured.

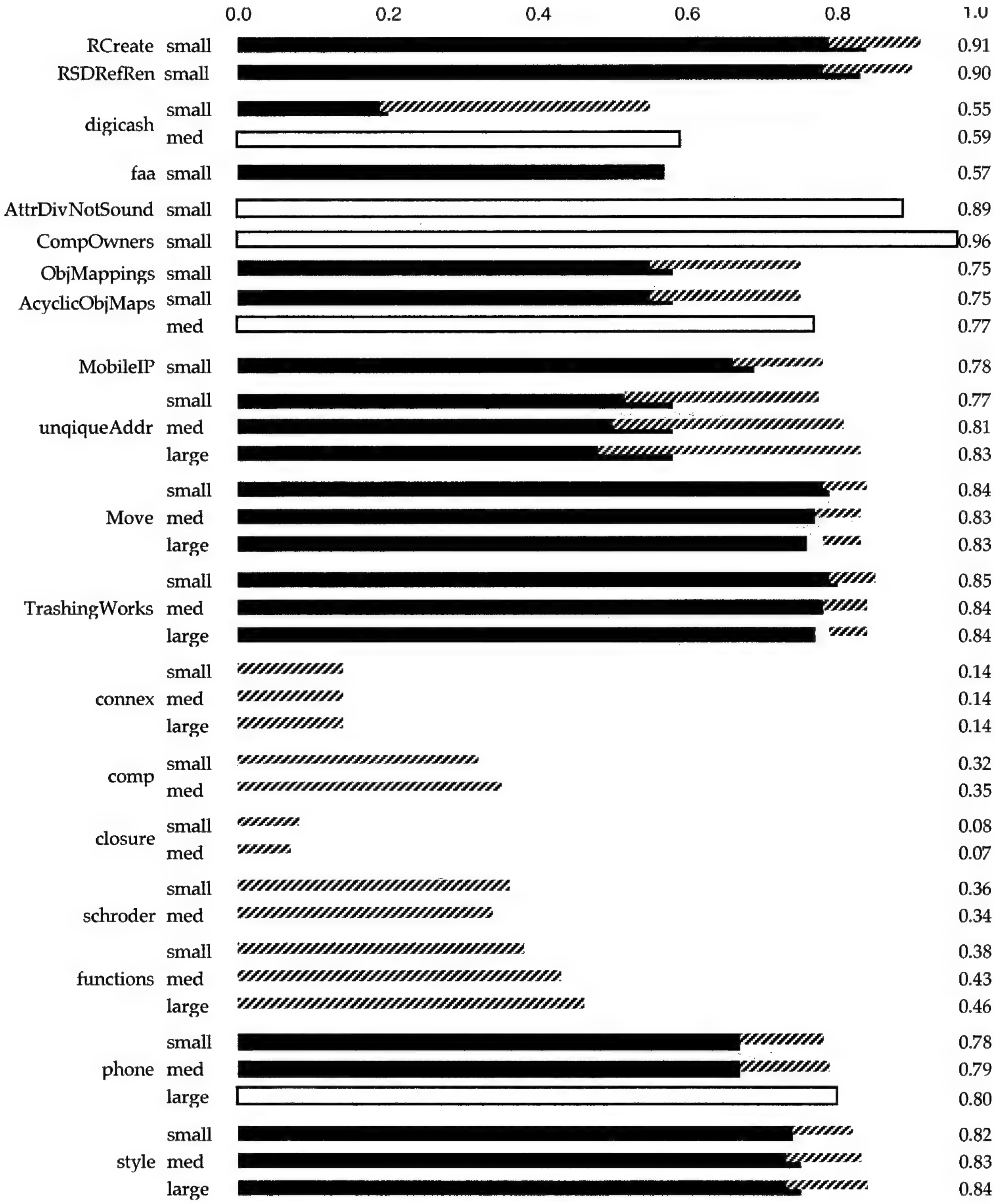


Figure 6.1. The reduction ratios for the benchmark suite. The solid bars represent the reduction from partial assignment techniques, the slash bars represent the reduction from isomorph elimination and the empty boxes are the total reduction where no breakdown is feasible.

Figure 6.1 shows that the partial-assignment reductions provide most of the reductions gained in Ladybug. At one extreme, the claims from the math specification have few constraints, all of which involve every variable, and offer no opportunities for the partial-assignment techniques. At the other extreme, the faa specification distinguishes the two controllers considered, allowing no isomorph elimination. Ladybug exploits both duplications to reduce all other tests. For most tests, the reduction ratio for the partial-assignment techniques ranges from six to eight times larger than the reduction ratio from isomorph elimination.

If the reduction gained from the partial-assignment techniques is orthogonal to the reduction gained from isomorph elimination, the slashed bar and the solid bar would exactly meet. In most cases, the slashed bar and the solid bar overlap. This overlap represents the additional reduction gained by the partial-assignment techniques or isomorph-elimination on the entire space of assignments instead of considering only the reduced space of assignments produced by the other approach. This overlap is significant in only a few of the tests, indicating that the two approaches interact well. In a few cases, the full reduction bar has a gap between the solid and slashed bars. This gap indicates that the isomorph-elimination reduction performs noticeably better for the values generated by the partial-assignment reductions than for those values removed by them.

Figure 6.1 also gives a first answer to the scaling question. For the tests with varying scopes, the reduction ratio remains roughly constant as the scope grows. In many of these cases, the reduction ratio actually grows slightly as the scope grows. The behavior of isomorph elimination can be seen most easily in the math claims. For relation values, the number of values grows exponentially with the square of the scope, whereas the reduction grows with the factorial of the scope. The actual reduction depends on the number of given types used in the search, but factorial grows more slowly than two to the square of the scope, so usage of relational values will generally grow poorly with scope. Function values, on the other hand, grow exponentially with the size of the scope. If multiple given types are available, the number of functions grows more slowly than the factorial growth of the isomorph-elimination reduction, leading to an improved reduction with the size of scope.

The success of the partial-assignment techniques depends on the filter formulae available and the variables being constrained. The logarithmic reduction ratio provided by some filter formulae grows quickly as the scope grows, whereas the ratio declines for other formulae. Likewise, more values can be reduced for some variables, such as relation-valued variables, than other variables, such as scalar-valued variables. These two factors determine the reduction ratio for bounded generation and derived variables.

Short circuiting introduces another complication. It does not prune values of the variable being constrained, but instead prunes subtrees rooted at those values; thus short circuiting is more effective for variables high in the search tree. This dependence on variable ordering may explain why previous search reductions based purely on backtracking scale poorly with scope.

Whereas the ratio is nearly constant across changing scopes for a given problem, the ratio generally rises with the size of the problem. Figure 6.2 lists the reduction ratios of the specifications that are drawn from real-world problems, listed in order of the size of the total search space. The upward trend in the ratio as the problem size grows is obvious in this figure. Two factors contribute to this positive relationship: more variables and more constraints. Although not necessary to increase the search space size, more variables are typically a major factor in the increased search space. Each additional variable gives additional chances for partial-assignment reductions. The second factor may be more psychological than causal; people appear to add more constraints to specifications with more variables. Each additional constraint adds additional opportunities for partial-assignment reductions.

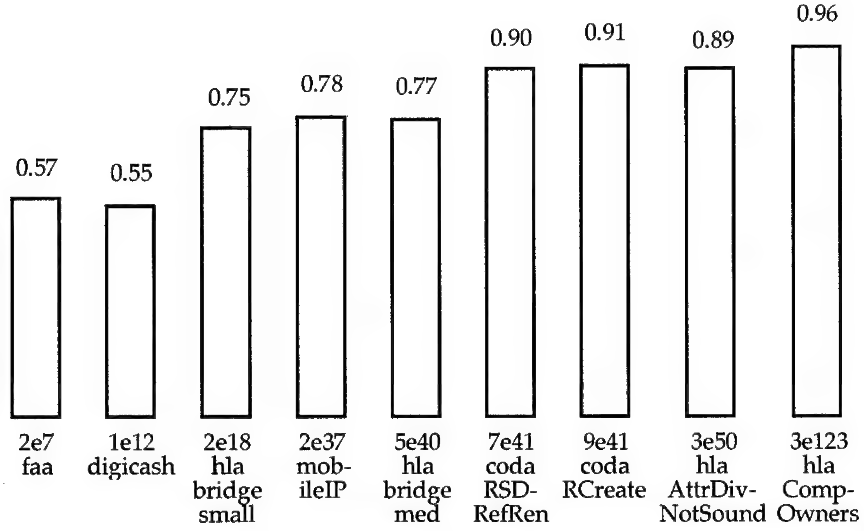


Figure 6.2. Reduction ratios for the "real-world" specifications, sorted by size. The size, in number of nodes in the search tree, is given immediately below each bar.

Ladybug cannot fully analyze every test in the suite in its entirety. The tests exceeding Ladybug's abilities fall into two categories: the math claims and some claims from "real-world" specifications. As noted previously, the math claims allow no reduction from the partial-assignment techniques, losing the most powerful techniques. The lack of constraints is an unrealistic part of the math claims; "real-world" software specifications generally include strong constraints.

Most of the "real-world" claims that are expensive or intractable to analyze fully return a counterexample quickly. In these cases, the full search discovers an enormous number of counterexamples. For the HLA AttrAcqNotSound claim, more than 95% of the values generated contribute to a counterexample that is generated. These counterexamples are nearly isomorph free, so Ladybug must generate a huge number of these values.

The notable exception to the quick analyses is the HLA bridge claims with the medium scope. The bridge claims exhibit a unique problem. Bounded generation cannot currently take advantage of the function composition or transitive closure operations; most of the constraints in the bridge claims involve one or both of these operations. The other operations generally use subset or intersection, both involving the identity function. The domain and range of the identity function is the entire given type, so no bounded generation reductions can use these constraints either.

An oddity worth noting occurs with the SpendOnce claim in digicash. Although Ladybug can discover a counterexample to the claim for the large scope in one second, it takes almost eleven minutes to find the corresponding counterexample for the medium scope. This anomaly is a result of the instability of the variable ordering heuristic. For the first clause of the normalized formula, Ladybug chooses a poor ordering for the medium scope, while choosing the same good ordering for both the small and large scopes. This clause, which is unsatisfiable, is expensive to analyze with the poor ordering.

6.3 Partial Assignment Results

This section considers the effectiveness of the partial-assignment techniques in more detail. The techniques are considered both in unison and separately. Two questions underlie this section:

Spec	Claim/ Operation	Scope	# Full Assigns	Assigns Checked	# Values	Values Generated	Value Reduction	Time to cover
coda	RCreate	small	9×10^{41}	0	1×10^{42}	6,493,752	0..84	6:24
	RSDRefRen	small	7×10^{41}	0	1×10^{42}	1×10^{7}	0.83	13:33
digicash	SpendOnce	small	1×10^{12}	1×10^{9}	1×10^{12}	1×10^{9}	0.25	37:01
		med	3×10^{20}	—	3×10^{20}	—	—	—
		large	3×10^{30}	—	6×10^{30}	—	—	—
faa	X1b_OK	small*	2×10^{7}	560	2×10^{7}	1,327	0.57	0
HLA owners	AttrDivNot	med*	3×10^{50}	—	3×10^{50}	—	—	—
	AttrAcqNot	med*	3×10^{50}	—	3×10^{50}	—	—	—
	CompOwners	med*	3×10^{123}	0	3×10^{123}	5×10^{8}	0.93	1:11:29
HLA bridge	ObjMapping	small*	2×10^{18}	4×10^{7}	2×10^{18}	7×10^{7}	0.57	12:25
	AcylicObjMaps	small*	2×10^{18}	7,478,859	2×10^{18}	1×10^{7}	0.62	2:28
	AcylicObjMaps	med*	5×10^{40}	—	5×10^{40}	—	—	—
mobile IP	loc_update_ok	small*	2×10^{37}	3×10^{8}	3×10^{37}	2×10^{10}	0.73	24:23:26
alloc	UniqueAddr	small	786,432	300	1,053,192	320	0.58	0
		med	4×10^{8}	4,320	5×10^{8}	4,368	0.57	0
		large	3×10^{11}	72,030	4×10^{11}	72,134	0.58	0
finder	Move	small	4×10^{14}	420	5×10^{14}	1,149	0.79	0
		med	7×10^{20}	36,792	9×10^{20}	60,541	0.77	1
		large	2×10^{27}	3,419,760	3×10^{27}	4,704,813	0.76	1:04
	TrashingWorks	small	4×10^{14}	180	5×10^{14}	903	0.80	0.
		med	7×10^{20}	14,424	9×10^{20}	35,869	0.78	0
		large	2×10^{27}	1,262,700	3×10^{27}	2,375,053	0.77	34
phone	CallersCalledP	small	4×10^{13}	27,648	6×10^{13}	32,268	0.67	0
		med	2×10^{23}	4×10^{7}	2×10^{23}	4×10^{7}	0.67	2:54
		large	2×10^{35}	—	3×10^{35}	—	—	—
styles	FormattingP	small	4×10^{18}	1,368	6×10^{18}	69,354	0.74	1
		med	1×10^{28}	266,268	1×10^{28}	1×10^{7}	0.75	6:12
		large	2×10^{38}	—	2×10^{38}	—	—	—

Table 6.5: Results of Ladybug checking the claims in the benchmark suite with all partial assignment techniques enabled and isomorph elimination disabled. The number of cases and values generated is the number required to cover the entire space. The claims from the math specification offer no opportunities for partial assignment reduction and have been omitted from this table.

- How effective is each technique in isolation?
- How much overlap exists between the values removed by the three techniques?

Table 6.5 shows the effectiveness of the combination of all partial-assignment techniques at reducing the search space. The claims from the math specification have been omitted, as none of those claims provide any opportunities for partial-assignment reduction.

The importance of isomorph elimination is immediately obvious when examining Table 6.5; even with the math claims omitted, the complete analysis of many claims is now intractable. If the

Specification	Claim/ Operation	Scope	# Full Assigns	Assigns Checked	# Values	Values Generated	Value Reduction
coda	RCreate	small	9×10^{41}	0	1×10^{42}	3×10^{9}	0.77
	RSDRefRen	small	7×10^{41}	0	1×10^{42}	6×10^{8}	0.79
digicash	SpendOnce	small	1×10^{12}	2×10^{10}	1×10^{12}	2×10^{10}	0.14
		med	3×10^{20}	—	3×10^{20}	—	—
		large	3×10^{30}	—	6×10^{30}	—	—
faa	X1b_OK	small*	2×10^{7}	280	2×10^{7}	12,988	0.44
HLA owners	AttrDivNot	small*	3×10^{50}	—	3×10^{50}	—	—
	AttrAcqNot	small*	3×10^{50}	—	3×10^{50}	—	—
	CompOwners	small*	3×10^{123}	0	3×10^{123}	729	0.98
HLA bridge	ObjMapping	small*	2×10^{18}	1×10^{9}	2×10^{18}	2×10^{9}	0.49
	AcylicObjMaps	small*	2×10^{18}	278,016	2×10^{18}	1,492,925	0.66
	AcylicObjMaps	med*	5×10^{40}	—	5×10^{40}	—	—
mobile IP	loc_update_ok	small*	2×10^{37}	—	3×10^{37}	—	—
alloc	UniqueAddr	small	786,432	9,216	1,053,192	10,107	0.33
		med	4×10^{8}	1,250,000	5×10^{8}	1,270,580	0.30
		large	3×10^{11}	3×10^{8}	4×10^{11}	3×10^{8}	0.27
finder	Move	small	4×10^{14}	2,322	5×10^{14}	404,264	0.62
		med	7×10^{20}	746,496	9×10^{20}	4×10^{7}	0.64
		large	2×10^{27}	3×10^{8}	3×10^{27}	4×10^{10}	0.61
	TrashingWorks	small	4×10^{14}	0	5×10^{14}	997,376	0.59
		med	7×10^{20}	5×10^{7}	9×10^{20}	2×10^{8}	0.60
		large	2×10^{27}	—	3×10^{27}	—	—
phone	CallersCalledP	small	4×10^{13}	4,018,176	6×10^{13}	4,757,196	0.51
		med	2×10^{23}	—	2×10^{23}	—	—
		large	2×10^{35}	—	3×10^{35}	—	—
styles	FormattingP	small	4×10^{18}	216,576	6×10^{18}	1,794,815	0.67
		med	1×10^{28}	4×10^{8}	1×10^{28}	3×10^{9}	0.66
		large	2×10^{38}	—	2×10^{38}	—	—

Table 6.6: Results of Ladybug checking the claims in the benchmark suite with only short circuiting enabled. The number of cases and values generated is the number required to cover the entire space.

search could not be completed within approximately 24 hours, the entry in the table is noted with a —. With the notable exception of the math specification, the partial-assignment techniques achieve a substantial reduction of their own, indicating that both approaches in combination are needed to make the searches tractable.

Partial-assignment reductions are directly tied to the opportunities for reductions presented by the atomic formulae. As shown in Figure 6.3, a clear correlation exists between the number of atomic formulae available per clause and the log of the reduction offered by short circuiting.

These atomic formulae represent the "ammunition" used by short circuiting, so a positive correlation is to be expected. The logarithmic relationship, however, provides evidence as to why

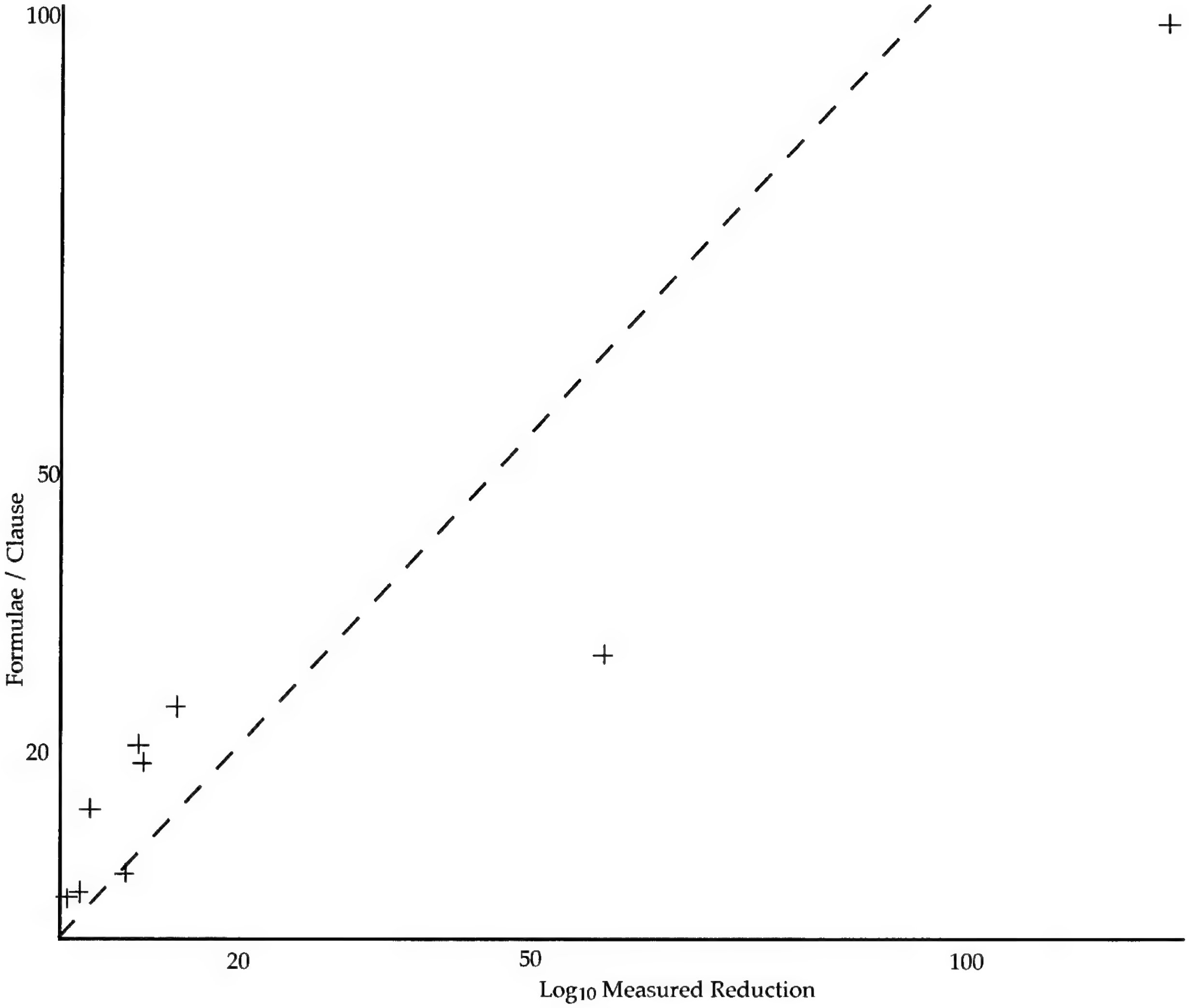


Figure 6.3. The reduction gained from short circuiting as related to the number of atomic formulae in the formula being solved.

tially related to the complexity of the number of variables, is inversely exponentially related to the complexity of the specification. The relational search problem is NP-complete, so no approach can be expected to provide a polynomial time solution for all formulae. Tricks like short circuiting, however, can reduce the exponent to make solvers tractable for many problems.

For derived variables, not all atomic formulae lead to increased reductions; only formulae that constrain the i^{th} variable to a single value support derived variables. Ladybug considers related formulae for derived variables only if they equate the i^{th} variable to a term involving only variables from Var_{i-1}.

Table 6.7 shows the number of variables that Ladybug derives for each claim in the benchmark suite. Most claims allow about half of the variables to be derived. The math claims are a notable exception; they are sufficiently succinct to disallow any derived variables. On the other hand, the coda claims and the CompleteOwners claim of the HLA owners specification allow about two thirds of its variables to be derived. This abundance of derived variables is common of claims,

Specification	Claim	Vars / Clause	Derived / Clause
coda	RCreate	33	22
	RSDRefRename	33	22
digicash	SpendOnce	6.6	0.2
faa	X1b_OK	10	4
HLA owners	AttrDivNot	27	13
	AttrAcqNot	27	13
	CompleteOwners	65	44
HLA bridge	Object Mapping	16	6
	AcyclicObjMaps	16	6
mobile IP	loc_update_ok	28	14
alloc	UniqueAddrAlloc	5	2
finder	Move	14	6
	TrashingWorks	14	6
math	connex	1	0
	comp	3	0
	closure	3	0
	schroder	2	0
	functions	3	0
phone	CallersCalledP	7	3
styles	FormattingP	13	4

Table 6.7: Availability of derived variables by claim.

such as these ones, that are built by composing a number of operations in sequence, each of which is described largely constructively.

The reduction for the derived-variable technique is easily predicted: it is always the product of the number of values possible for each variable that can be derived. Although the reductions are significant in most cases, the analysis of many claims remains intractable, including most from the realistically sized specifications. Derived variable generation helps, but by itself it is insufficient to allow the analysis of interesting claims. Table 6.8 shows the computed results of using only derived variable reduction in performing the searches entailed by the benchmark suite. Most of these searches are infeasible to perform.

Table 6.9 gives the results of using only bounded generation to reduce the size of the search for the benchmark suite. Comparing Table 6.9 to Table 6.6, fewer of the bounded-generation searches are feasible than are feasible using only short circuiting. Short circuiting has an advantage in that it can exploit more atomic formulae and is thus successfully applicable to a larger set of searches.

On the other hand, bounded generation by itself requires fewer assignments to be generated

Specification	Claim/ Operation	Scope	# Full Assigns	Assigns Checked	# Values	Values Generated	Value Reduction
coda	RCreate	small	9×10^{41}	1×10^{15}	1×10^{42}	2×10^{15}	0.64
	RSDRefRen	small	7×10^{41}	6×10^{14}	1×10^{42}	9×10^{14}	0.64
digicash	SpendOnce	small	1×10^{12}	2×10^{11}	1×10^{12}	2×10^{11}	0.06
		med	3×10^{20}	3×10^{18}	3×10^{20}	3×10^{18}	0.10
		large	3×10^{30}	1×10^{27}	6×10^{30}	1×10^{27}	0.12
faa	X1b_OK	small*	2×10^{7}	4,096	2×10^{7}	5,460	0.49
HLA owners	AttrDivNot	small*	3×10^{50}	2×10^{22}	3×10^{50}	2×10^{22}	0.56
	AttrAcqNot	small*	3×10^{50}	2×10^{22}	3×10^{50}	2×10^{22}	0.56
	CompOwners	small*	3×10^{123}	5×10^{38}	3×10^{123}	5×10^{38}	0.69
HLA bridge	ObjMapping	small*	2×10^{18}	2×10^{11}	2×10^{18}	2×10^{11}	0.38
	AcylicObjMaps	small*	2×10^{18}	2×10^{11}	2×10^{18}	2×10^{11}	0.38
	AcylicObjMaps	med*	5×10^{40}	1×10^{31}	5×10^{40}	1×10^{31}	0.24
mobile IP	loc_update_ok	small*	2×10^{37}	1×10^{17}	3×10^{37}	2×10^{17}	0.54
alloc	UniqueAddr	small	786,432	1,536	1,053,192	2,056	0.45
		med	4×10^{8}	40,000	5×10^{8}	50,016	0.46
		large	3×10^{11}	1,244,160	4×10^{11}	1,493,024	0.47
finder	Move	small	4×10^{14}	2×10^{7}	5×10^{14}	3×10^{7}	0.49
		med	7×10^{20}	3×10^{10}	9×10^{20}	3×10^{10}	0.50
		large	2×10^{27}	4×10^{13}	3×10^{27}	5×10^{13}	0.50
	TrashingWorks	small	4×10^{14}	2×10^{7}	5×10^{14}	3×10^{7}	0.49
		med	7×10^{20}	3×10^{10}	9×10^{20}	3×10^{10}	0.50
		large	2×10^{27}	4×10^{13}	3×10^{27}	5×10^{13}	0.50
phone	CallersCalledP	small	4×10^{13}	294,912	6×10^{13}	295,500	0.60
		med	2×10^{23}	7×10^{8}	2×10^{23}	7×10^{8}	0.62
		large	2×10^{35}	1×10^{12}	3×10^{35}	1×10^{12}	0.66
styles	FormattingP	small	4×10^{18}	9×10^{10}	6×10^{18}	9×10^{10}	0.42
		med	1×10^{28}	2×10^{16}	1×10^{28}	2×10^{16}	0.42
		large	2×10^{38}	2×10^{22}	2×10^{38}	2×10^{22}	0.42

Table 6.8: The computed results of Ladybug checking the claims in the benchmark suite with only derived variables enabled. The number of cases and values generated is the number required to cover the entire space.

than short circuiting for most of the tractable searches. Because it can exploit the same atomic formulae, for any ordering of the variables, short circuiting by itself will yield no more assignments at any level (below *N*) than bounded generation, and generally fewer. But short circuiting will generate at least as many values and generally more.

All three techniques largely remove the same full assignments from the search space. Short circuiting removes every full assignment removed by the other two techniques and generally removes full assignments not removed by the others. The other techniques, on the other hand, remove many *partial* assignments left by short circuiting.

Table 6.10 lists the reduction ratios for the claims in the benchmarks that were tractable (or

Specification	Claim/ Operation	Scope	# Full Assigns	Assigns Checked	# Values	Values Generated	Value Reduction
coda	RCreate	small	9×10^{41}	—	1×10^{42}	—	—
	RSDRefRen	small	7×10^{41}	—	1×10^{42}	—	—
digicash	SpendOnce	small	1×10^{12}	—	1×10^{12}	—	—
		med	3×10^{20}	—	3×10^{20}	—	—
		large	3×10^{30}	—	6×10^{30}	—	—
faa	X1b_OK	small*	2×10^{7}	1,920	2×10^{7}	21,682	0.41
HLA owners	AttrDivNot	small*	3×10^{50}	—	3×10^{50}	—	—
	AttrAcqNot	small*	3×10^{50}	—	3×10^{50}	—	—
	CompOwners	small*	3×10^{123}	—	3×10^{123}	—	—
HLA bridge	ObjMapping	small*	2×10^{18}	—	2×10^{18}	—	—
	AcylicObjMaps	small*	2×10^{18}	—	2×10^{18}	—	—
	AcylicObjMaps	med*	5×10^{40}	—	5×10^{40}	—	—
mobile IP	loc_update_ok	small*	2×10^{37}	—	3×10^{37}	—	—
alloc	UniqueAddr	small	786,432	972	1,053,192	2,006	0.45
		med	4×10^{8}	26,620	5×10^{8}	53,445	0.46
		large	3×10^{11}	856,830	4×10^{11}	1,714,340	0.46
finder	Move	small	4×10^{14}	37,248	5×10^{14}	71,984	0.67
		med	7×10^{20}	7×10^{7}	9×10^{20}	1×10^{8}	0.62
		large	2×10^{27}	—	3×10^{27}	—	—
	TrashingWorks	small	4×10^{14}	20,160	5×10^{14}	45,998	0.68
		med	7×10^{20}	3×10^{7}	9×10^{20}	3×10^{7}	0.62
		large	2×10^{27}	—	3×10^{27}	—	—
phone	CallersCalledP	small	4×10^{13}	—	6×10^{13}	—	—
		med	2×10^{23}	—	2×10^{23}	—	—
		large	2×10^{35}	—	3×10^{35}	—	—
styles	FormattingP	small	4×10^{18}	1×10^{8}	6×10^{18}	2×10^{8}	0.56
		med	1×10^{28}	—	1×10^{28}	—	—
		large	2×10^{38}	—	2×10^{38}	—	—

Table 6.9: Results of Ladybug checking the claims in the benchmark suite with only bounded generation enabled. The number of cases and values generated is the number required to cover the entire space.

predictable) for at least two of the partial-assignment techniques in isolation. With two exceptions, the combination of the techniques outperformed any technique individually, significantly in almost every case. However, the combination is far less effective than would be expected if the techniques were independent. This observation indicates the expected interaction: all three techniques remove many of the same cases.

The generally significant improvement from the combination is also to be expected. Derived variables and bounded generation prevent the generation of values that the others will require. Short circuiting can take advantage of atomic formulae that the other two cannot. A second feature of the techniques is also notable in this table; each technique in isolation is the "best" technique of

Specification	Claim	Scope	Derived Variable Reduction	Short Circuiting Reduction	Bounded Gen Reduction	Partial Assgnmnt Reduction
alloc	UniqueAddr	small	0.45	0.36	0.45	0.58
		med	0.46	0.30	0.46	0.58
		large	0.47	0.27	0.47	0.58
coda	RCreate	small	0.64	0.77	—	0.84
	RSDRefRen	small	0.64	0.79	—	0.83
digicash	SpendOnce	small	0.06	0.14	—	0.25
faa	X1b_OK	small	0.49	0.44	0.41	0.57
finder	Move	small	0.49	0.62	0.67	0.79
		med	0.5	0.64	0.62	0.77
	TrashingWorks	small	0.49	0.59	0.68	0.80
		med	0.5	0.60	0.64	0.78
HLA owners	CompOwners	small	0.69	0.98	—	0.93
HLA bridge	ObjMapping	small	0.38	0.49	—	0.57
	AcyclicObjMaps	small	0.38	0.66	—	0.62
phone	CallersCalledP	small	0.60	0.48	—	0.67
styles	FormattingP	small	0.42	0.67	0.56	0.74
		med	0.61	0.66	—	0.75

Table 6.10: The reduction ratios for selected searches from the benchmark suite with the partial-assignment techniques enabled individually or in combination.

at least two tests and is the "worst" technique on at least one test. This contrast allows the combination to perform acceptably over a larger range of formulae.

The oddity of having short circuiting alone outperform the combination for two tests (CompOwners and AcyclicObjMaps) is again the result of inaccuracies in the ordering heuristic.

6.4 Isomorph Elimination

The previous sections make it clear that isomorph elimination is an important reduction for Ladybug. The traditional problem with isomorph elimination is its cost; most search implementations have a high cost for isomorph elimination that grows quickly with the size of the scope. This section focuses on the cost of the isomorph-eliminating generators. The first comparison is between the time per value generated for isomorph-eliminating generators and for exhaustive generators. The second comparison looks at the change in the cost of isomorph-eliminating generators as the scope grows.

In Ladybug, the cost of the generators cannot easily be separated from the cost for the tests, as each value is both generated and tested. However, the test costs are independent of the generator

Specification	Claim/ Operation	Scope	Values W Isomorph	Time W Isomorph	μs per Value	Vals W/O Isomorph	Time W/O Isomorph	μs per Value
coda	RCreate	small	8,788	1	113	6,493,752	6:24	59
	RSDRefRen	small	17,842	1	56	1×10^7	13:33	81
digicash	SpendOnce	small	288,323	4	14	1×10^9	36:28	2
		med	3×10^8	1:47:14	21	—	—	—
HLA owners	AttrDivNot	med*	240,614	3	12	—	—	—
	CompOwners	med*	252,330	3	12	5×10^8	1:11:29	9
HLA bridge	ObjMapping	small*	42,937	1	23	7×10^7	12:25	11
	AcyclicObjMaps	small*	42,643	1	23	1×10^7	2:28	15
		med*	2×10^9	21:15:08	38	—	—	—
math	connex	large	5,538,962	29	5	7×10^7	3:19	3
	comp	small	284,467	4	14	—	—	—
		med	3×10^9	8:02:19	10	—	—	—
	closure	small	91,486	1	11	—	—	—
		med	8×10^8	4:27:03	20	—	—	—
	shroder	small	257,766	6	23	—	—	—
		med	5×10^9	21:31:52	16	—	—	—
	functions	small	2,285	1	438	—	—	—
		med	58,667	1	17	—	—	—
		large	1,845,962	54	29	—	—	—
mobile IP	loc_update_ok	small*	2×10^8	1:34:12	28	3×10^{11}	24:23:26	0.3
phone	CallersCalledP	med	68,648	1	15	4×10^7	2:54	4
		large	1×10^7	2:19	14	—	—	—
styles	FormattingP	med	63,686	4	63	1×10^7	6:12	37
		large	1,850,891	1:49	59	—	—	—

Table 6.11: Results of Ladybug checking selected claims in the benchmark suite with only isomorph elimination enabled. The number of cases and values generated is the number required to cover the entire space.

used. Limited investigations also indicate that the tests are slightly less than half of the total cost of the search.

Table 6.11 compares the search times using isomorph-eliminating and exhaustive generators for all tractable searches that required at least one second to complete. A few of the numbers for the searches requiring only a second appear to be skewed; I expect a small, more constant factor becomes dominant in some cases.

Considering the first question, the cost of the isomorph-eliminating generators appears to be relatively minor, roughly doubling the cost per value for the typical test. This increase in the cost is more than compensated for by the decrease in the number of values being generated.

The second question is the growth in the cost of generation as the scope grows. The generation algorithm is limited to cubic in the scope, but cubic growth would present problems for even moderate scopes. Although the cost of generation frequently does grow with the scope, the growth is moderate. In several cases, the cost of generation per value actually drops as the scope grows. For at least the moderate-sized scopes considered in Ladybug, the algorithms implemented for Ladybug's isomorph-eliminating generators are well-behaved and do not become exceedingly expensive.

6.5 Conclusions

The empirical data presented in this chapter support two broad conclusions about selective enumeration, as implemented in Ladybug. Selective enumeration performs well enough across a variety of claims and specifications to make those claims checkable. Although still exponential in the size of the scope, selective enumeration scales well in scope and size of specification.

However, several caveats can also be gained from the data. Ladybug requires a rich specification with a broad set of constraints for large searches to be reduced to a tractable size. The math claims are the most notable example of a specification that lacks these constraints.

Several anomalies occurred due to the approximations used by the variable ordering heuristic causes. This opens the questions as to whether some of the intractable searches may have been possible with a better variable ordering. Revisiting the variable ordering heuristic is probably warranted.

Finally, some operators disallow bounded generation, significantly harming the potential for reducing the size of the search. This restriction was most notable in hla bridge claims, where the primary tests involved testing for cycles in composed relations. Finding ways to improve the reductions for these operators is also a worthy target for future consideration.

Chapter 7

Analyzing HLA

This chapter develops a case study that shows how Ladybug discovered flaws in a significant "real-world" specification. The High Level Architecture (HLA) specification describes a set of protocols for a distributed simulation environment. Ladybug, with its strengths in analyzing structural properties, is the appropriate tool for analyzing some of the properties of the HLA, notably properties about the data structures maintained in the environment. Other tools, such as model checkers, are more appropriate for other properties, such as potential deadlocks or race conditions.

Regardless of the tool chosen, formal analysis of a specification requires three basic steps: choosing the facets of the system to be analyzed, modeling the appropriate portions of the system, and analyzing that model with the chosen tool. This case study analyzes two portions of the complete HLA specification: the ownership management services and the topologies of federations allowed with bridges. Several people, including myself, worked on modeling and analyzing the ownership management services; the work is originally reported in [DM+99].

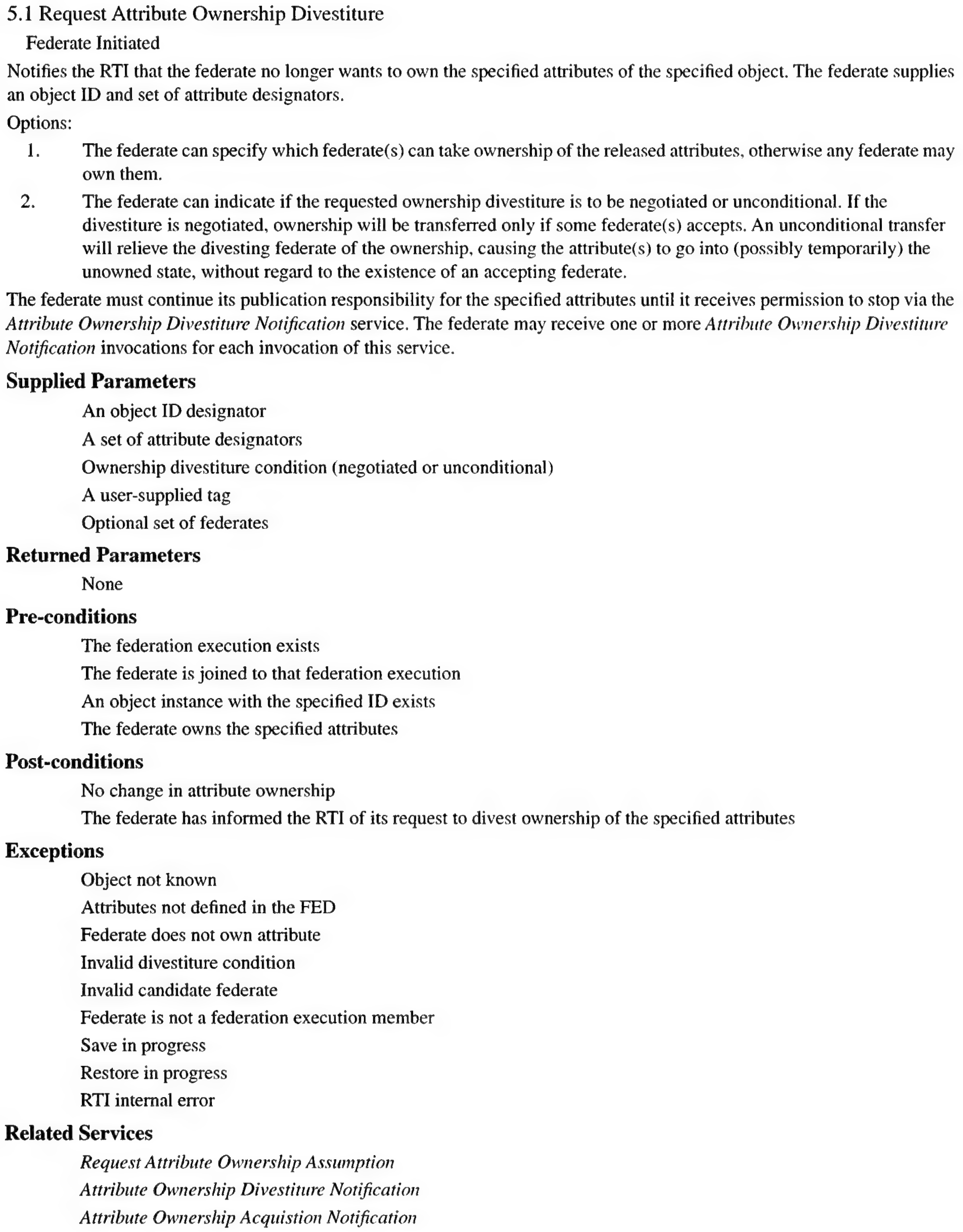
The first section of this chapter provides background information on the HLA specification. Section 7.2 describes the formal model of the HLA, expressed in the specification language Z. Section 7.3 describes the analysis of this model, including both the translation to NP and the usage of Ladybug. Section 7.4 discusses what was learned in this case study.

7.1 Overview of HLA

Beginning in 1996, the Defense Modeling and Simulation Office (DMSO) of the United States Department of Defense developed a component integration standard for distributed simulation called the "High Level Architecture" (HLA). Informally, the HLA prescribes a kind of "simulation bus" into which simulations can be "plugged" to produce a joint (distributed) simulation. A goal of the standard was to allow independent vendors to develop simulations that can be combined for use in a unified simulation with minimal complications.

In the HLA design, members of a *federation* — the HLA term for a distributed simulation — coordinate their models of parts of the world by sharing objects of interest and the attributes that define them. Each member of the federation is called a *federate*. A federate is responsible for calculating some part of the larger simulation and broadcasting updates using the facilities of the runtime infrastructure, termed the *RTI*.

The "Interface Specification" document or *IFSpec* [DoD97] defines routines that support com-



5.1 Request Attribute Ownership Divestiture

Federate Initiated

Notifies the RTI that the federate no longer wants to own the specified attributes of the specified object. The federate supplies an object ID and set of attribute designators.

Options:

1. The federate can specify which federate(s) can take ownership of the released attributes, otherwise any federate may own them.
2. The federate can indicate if the requested ownership divestiture is to be negotiated or unconditional. If the divestiture is negotiated, ownership will be transferred only if some federate(s) accepts. An unconditional transfer will relieve the divesting federate of the ownership, causing the attribute(s) to go into (possibly temporarily) the unowned state, without regard to the existence of an accepting federate.

The federate must continue its publication responsibility for the specified attributes until it receives permission to stop via the *Attribute Ownership Divestiture Notification* service. The federate may receive one or more *Attribute Ownership Divestiture Notification* invocations for each invocation of this service.

Supplied Parameters

An object ID designator

A set of attribute designators

Ownership divestiture condition (negotiated or unconditional)

A user-supplied tag

Optional set of federates

Returned Parameters

None

Pre-conditions

The federation execution exists

The federate is joined to that federation execution

An object instance with the specified ID exists

The federate owns the specified attributes

Post-conditions

No change in attribute ownership

The federate has informed the RTI of its request to divest ownership of the specified attributes

Exceptions

Object not known

Attributes not defined in the FED

Federate does not own attribute

Invalid divestiture condition

Invalid candidate federate

Federate is not a federation execution member

Save in progress

Restore in progress

RTI internal error

Related Services

Request Attribute Ownership Assumption

Attribute Ownership Divestiture Notification

Attribute Ownership Acquistion Notification

Figure 7.1. The Request Attribute Ownership Divestiture service of the RTI, as specified in the IFSpec [DoD97] (version 1.2).

munication both from the federates (e.g., to indicate new values) and to the federates (e.g., to request updates for a particular attribute). The IFSpec defines routines, or "services", by a name, the initiator (either a federate or the RTI), a set of parameters, a possible return value, pre- and post-conditions, and a list of exceptions that may occur as a result of executing the service.

Figure 7.1 (taken from [DoD97]) shows an example of a typical RTI service. A federate initiates this service when it wants to relinquish ownership of some attributes of a particular object being simulated by the federate. The federate relinquishes ownership, however, only when informed by the RTI using the *Attribute Divestiture Notification* service.

The HLA is a complex integration framework. The current IFSpec includes over 125 different services, and the full document is over 400 pages of description. While the part of the HLA design that deals with attribute broadcast is relatively straightforward, the overall framework is complicated significantly by the need to deal with issues such as starting, stopping, and pausing; allowing one federate to transfer object ownership to another; and distributed clock management and time-ordered message sequencing.

To make the integration framework manageable, the IFSpec is divided into six chapters: federation management, declaration management, object management, ownership management, time management, and data distribution management. Federates use the federation management services to initiate a federation execution, to join or leave an execution in progress, to pause and resume, and to save execution state. Declaration management services communicate what kinds of object attributes are available and of interest, whereas object management services communicate actual object values. Ownership management services allow responsibility for calculating the value of an attribute to be transferred from one federate to another. Time management services coordinate the logical time advancements of federates and ensure that messages are delivered in time-stamp order. Data distribution management services filter attribute updates for each federate based on defined criteria, reducing message traffic and processing requirements.

7.1.1 The HLA Model of Attribute Ownership

Much of this case study focuses on the ownership management services, which control the transfer of ownership of attributes. The HLA adopts an object view of a distributed simulation; the simulation universe consists of a collection of objects, each of which has a set of attributes. The job of the overall simulation is to calculate and update values of these attributes over time. Different federates can calculate values of different attributes of the same underlying object.

Every object in an HLA simulation is an instance of some object class.[1] These classes define the attributes for their instances; the IFSpec defines an attribute to be "a distinct, identifiable portion of the object state". Version 1.2 of the IFSpec is at times inconsistent in its usage of the term attribute, sometimes meaning a value associated with a single object and at other times meaning the description of the state of all objects of a particular class. I use the phrase *object attribute* to describe the former case and *class attribute* to describe the latter case.[2] Comparing this to a language such as Java, the object attribute is the equivalent of the value of a field for a specific object and the class attribute is the field object itself, accessible only through the reflection mechanism in Java.

The IFSpec defines the object classes to support inheritance, which introduces some additional

[1] The IFSpec discusses two kinds of classes: object classes and interaction classes. Interaction classes are irrelevant to the work presented here, so I use the word class to refer to object classes.

[2] Partially in response to our concerns about this distinction, later versions of the IFSpec consistently distinguish object attributes from class attributes. They chose, however, to use the term instance attribute rather than object attribute.

Updates A federate updates an object attribute when it sends out a new value for the attribute. An update is an event, not a state.

Reflects A federate reflects an object attribute when it receives a new value for the attribute. A reflection is also an event, not a state.

Owns A federate owns an object attribute if it has the privilege to update values for that attribute. An object attribute should have no more than one owner at a time. Ownership is a state.

Publishes A federate publishes a class attribute if it could provide updates for that kind of attribute, whether or not it currently has the privilege to do so for any particular object. Publishing is a state, not a point event. Multiple federates may publish the same class attribute at the same time.

Subscribes A federate subscribes to a class attribute if it wants to receive updates to that kind of attribute. Subscribing is a state.

Figure 7.2. Brief glossary of IFSpec terms

complexity. None of the properties I model depend in any way upon this inheritance, so I drop consideration of inheritance. The choice of which portions of the model to consider and which to ignore is an important aspect of modeling a system for analysis.

The HLA propagates object attribute updates using a publish and subscribe system, although the IFSpec uses the standard terminology in a somewhat non-standard way. Figure 7.2 presents a brief glossary of the terminology (as used in the IFSpec).

As an overview, an object attribute of a given object is updated only if some federate

- *publishes* the corresponding class attribute,
- *owns* the attribute for that object, and
- *updates* the value

Another federate "sees" the updated value for the object attribute only if it *subscribes* to the corresponding class attribute. The receipt of this new value is known as a *reflection*.

A federate may own an object attribute only if it publishes the corresponding class attribute and no other federate owns that object attribute. A federate begins the acquisition of ownership of an object attribute by requesting ownership from the RTI using the *Request Attribute Ownership Acquisition* service. The RTI will respond, if possible, by granting ownership using the *Attribute Ownership Acquisition Notification* service.

A federate can similarly disown an object attribute by initiating the *Request Attribute Ownership Divestiture* service and waiting for the corresponding *Attribute Ownership Divestiture Notification* service invocation from the RTI. The divestiture request can either be unconditional, leading to a possibly unowned object attribute (which will therefore not be updated until another federate claims ownership), or it can be negotiated, with the federate maintaining ownership until the RTI can locate another federate willing to own the object attribute.

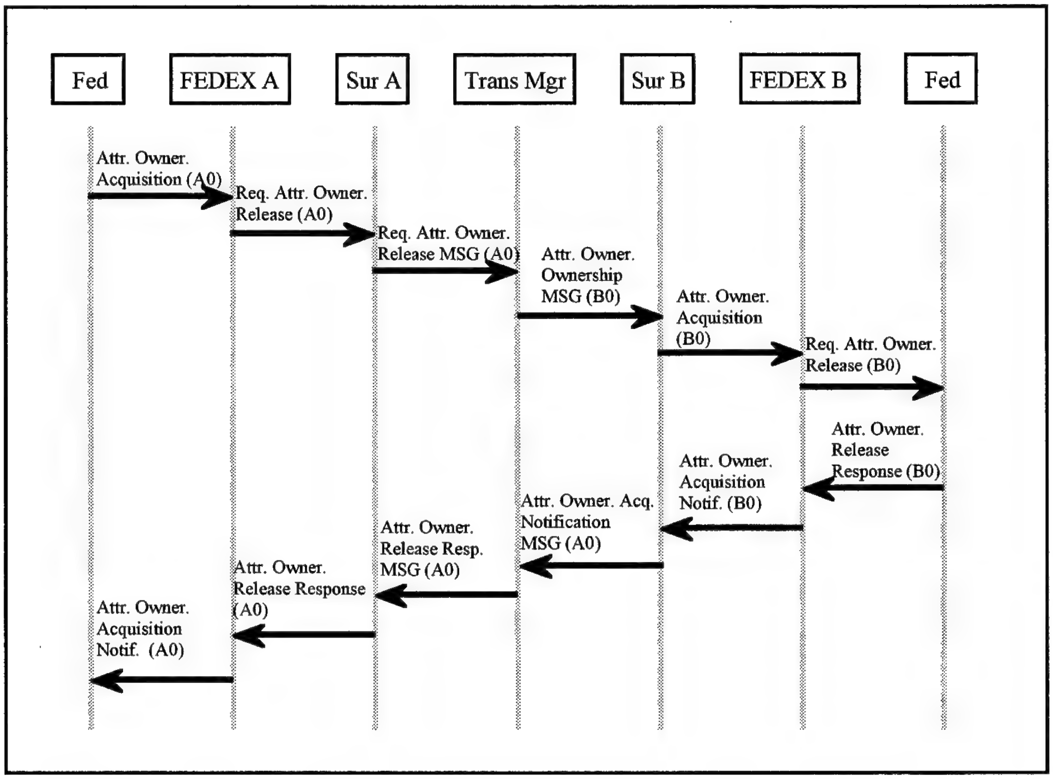
The RTI searches for possible owners of an object attribute that is being divested by invoking the *Request Attribute Ownership Assumption* service on federates that are currently publishing the corresponding class attributes. An interested federate may then be granted ownership of the object attribute by the RTI.

7.1.2 The HLA Bridge Concept

After the basic HLA system had begun to stabilize, members of the HLA community advanced the concept of bridging multiple federations as a way to improve the distribution opportunities

2.4.2 Attribute Ownership Acquisition Across the Bridge Federate

Federate A1 in Federation FEDEX A requests an *Attribute Ownership Acquisition*, which will be transmitted to each federation linked to the Bridge Federate.


Fed
FEDEX A
Sur A
Trans Mgr
Sur B
FEDEX B
Fed
Attr. Owner. Acquisition (A0)
Req. Attr. Owner. Release (A0)
Req. Attr. Owner. Release MSG (A0)
Attr. Owner. Ownership MSG (B0)
Attr. Owner. Acquisition (B0)
Req. Attr. Owner. Release (B0)
Attr. Owner. Release Response (B0)
Attr. Owner. Acquisition Notif. (B0)
Attr. Owner. Acq. Notification MSG (A0)
Attr. Owner. Release Resp. MSG (A0)
Attr. Owner. Release Response (A0)
Attr. Owner. Acquisition Notif. (A0)

1. Federate A1 issues an *Attribute Ownership Acquisition* for object A0 attribute A0.
2. SUR A receives a *Request Attribute Ownership Release* from FEDEX A for the object A0 attribute A0.
3. SUR A issues a *Request Attribute Ownership Release* message to the Transformation Manager.
4. The Transformation Manager issues an *Attribute Ownership Acquisition* message to SUR B for object B0 attribute B0 corresponding to the object A0 attribute A0 according to the FOMAT.
5. SUR B issues an *Attribute Ownership Acquisition* for object B0 attribute B0.
6. Federate B1 receives a *Request Attribute Ownership Release* for object B0 attribute B0.
7. Federate B1 issues an *Attribute Ownership Release Response* for object B0 attribute B0.
8. SUR B receives an *Attribute Ownership Acquisition Notification* indicating the object attributes that has been released.

(remaining steps elided)

Figure 7.3. The description of the attribute ownership aquisition protocol across a bridge, as given by Shen et.al. in [SB+98].

[SB+98]. Two or more federations can be joined by a special entity, called a *bridge federate*. A bridge federate consists of three kinds of components: two or more surrogate federates, a transformation manager, and a federation object management mapping/transformation specification (FOMAT). Surrogate federates join each federation being linked, appearing as normal federates to the other participants in the federation. The FOMAT contains the mappings of objects, classes, and attributes between the different federations. The object, attribute, and class identities are inconsistent across federations; the transformation manager is responsible for mapping these identities as it transforms services forwarded by one surrogate into the appropriate outgoing messages for the other surrogates.

The goal of the bridge concept is to allow multiple federations (sometimes called federation executions or FEDEX's) to interact without adding any new services to the HLA framework. Unlike the IFSpec, the bridge documentation does not define services. Instead, it provides message sequence charts [ITU93] for possible executions of many interesting protocols. Figure 7.3 shows the message sequence chart for the attribute ownership acquisition protocol across a bridge along with the beginning of the accompanying text. The remaining text, which is not included in Figure 7.3, describes the last few messages at a similar level of detail.

In every protocol, the processing is basically the same. The RTI believes that the surrogate that represents the true target federate is actually the target. The surrogate passes the service request to the appropriate surrogate, after being transformed by the transformation manager according to the mappings in the FOMAT. The second surrogate then behaves as the original requestor in the other federation, requesting the desired service from the RTI, which in turn makes the appropriate request to the eventual target. Nothing prevents this target federate from itself being a surrogate for a federate in yet another federation.

7.1.3 An Overview of the Analysis

Identifying the questions of the system to be studied is the first issue. For the ownership management services, several questions are worth considering:

- Can more than one federate gain ownership of a single object attribute at the same time?
- Can a federate gain ownership of an object attribute without publishing the corresponding class attribute?
- Does the negotiated divestiture protocol guarantee that all object attributes remain owned?

Several more questions relate specifically to bridges:

- Can an object appear more than once in a single federation?
- Can cycles arise in the mapping of objects?

Next, I describe the requirements of the ownership management section of the HLA specification using Z [Spi92]. I model only those portions of the ownership management that relate to the questions above. I similarly describe an extended model of the HLA that includes bridges, also using Z.

Special care must be taken when formalizing a specification for analysis. The simplest translation may prevent the discovery of some interesting flaws. To be checkable, the specification must clearly delineate those properties that are guaranteed to be true from those properties that are desired to be true.

Consider the ownership relationship from HLA as an example. We can model this relationship in Z as a relation mapping federates to object attributes, with each federate-attribute pair describing the ownership of a single object attribute by a federate. However, only a single federate is allowed to own a given object attribute at a time. We can encode this constraint in the Z description by making the relation injective. However, placing this encoding on the basic description of the system prevents Ladybug from discovering possible corruptions of the system involving

$[CLASS, CLASSATTR]$

$$\begin{array}{|l} AttributesToClass : CLASSATTR \rightarrow CLASS \\ privToDeleteObject : CLASS \rightarrowtail CLASSATTR \\ \hline privToDeleteObject\sim \subseteq AttributesToClass \end{array}$$

Figure 7.4. Z model of classes and class attributes

simultaneous owners. Instead, I encode this constraint as a separate property, allowing it to be checked. (See the description of *NoTwoOwners* in the next section for more details.)

Checking that the *Attribute Ownership Divestiture Notification* service maintains this property requires checking the claim

$$\mathsf{NoTwoOwners} \ \wedge \ \mathsf{AttrOwnDivestNotify} \ \Rightarrow \ \mathsf{NoTwoOwners}'$$

This formula says that if NoTwoOwners holds initially and AttrOwnDivestNotify is executed, then NoTwoOwners will still hold afterwards. If NoTwoOwners is not invariant across AttrOwnDivestNotify, Ladybug will provide concrete counterexamples demonstrating the violation of NoTwoOwners.

I then translate the Z notation to NP [JD96a], the input language used by Ladybug. NP is essentially a first-order subset of Z with the many special characters used in Z remapped to ASCII equivalents. In some instances, I must replace a Z construct that is not directly supported by NP with a less elegant or more complicated construct that is supported. I also encode the questions about the specification as claims in NP.

The final step before running Ladybug is to choose a scope. As described in Section 7.3.3, the formal model place some constraints on the scope. For example, the number of object attributes must be exactly the number of objects times the number of class attributes. I choose the scope that requires the fewest number of objects that seems likely to contain an error.

Running Ladybug is the final, and simplest, step. Section 7.3.3 describes the behavior of Ladybug on the two portions of this case study.

7.2 The Formal Model

This section describes a formal model of HLA, based on version 1.2 of the IFSpec. To reduce the burden on the reader, this section details only a representative sampling of properties and operations. The remaining properties and operations are similar to those described here. Appendix D gives the complete Z model developed.

I present the model in two distinct phases. The first phase models the ownership management services and consists of four major pieces:

- classes, objects, and attributes, which are global to all of HLA
- the state required (explicitly or implicitly) by the ownership management specification
- properties about the state
- operations on the state

The second phase extends the first two pieces of this model to consider multiple federations and bridges. Only minimal changes are required to the properties and operations. This phase also describes two additional properties.

```
[OBJECT, OBJECTATTR]

┌─ObjectCollection ──────────────────────────────────────────
│ Objects : ℙ OBJECT
│ ObjectsToClass : OBJECT → CLASS
│ ObjectAttrs : ℙ OBJECTATTR
│ ObjectAttrToObject: OBJECTATTR ↠ OBJECT
│ ObjectAttrToClassAttr : OBJECTATTR → CLASSATTR
├──────────────────
│ ObjectAttrs = dom (ObjectAttrToObject ▷ Objects)
│ ObjectToClass ; AttributesToClass~ = ObjectAttrToObject~ ; ObjectAttrToClassAttr
│ ( ObjectAttrToObject ; ObjectAttrToObject~) ∩
│                ( ObjectAttrToClassAttr ; ObjectAttrToClassAttr~)
│        ⊆ id OBJECTATTR
└────────────────────────────────────────────────────────────
```

Figure 7.5. Z model of objects and object attributes

7.2.1 Classes, Objects, and Attributes

Figure 7.4 gives the Z model of HLA classes and class attributes that will be used as the basis for all further descriptions. The first line introduces two basic kinds of entities: classes and class attributes. The axiomatic definition describes two functions and a constraint between them that must always be maintained. The *AttributesToClass* function maps every class attribute to a single class.

Within the HLA, the authority to delete an object is obtained by becoming the owner of the special attribute, called the *privilegeToDeleteObject* attribute, for that object. Although *privilegeToDeleteObject* is a fully defined class attribute, it is expected that federates will rarely, if ever, associate a value with it. This special attribute, which must be defined for every class, is modeled by the *privToDeleteObject* function in the Z model. The constraint requires that the privilege to delete attribute that is specially denoted for a class must be a class attribute for that class.

Figure 7.5 gives the initial description of objects in HLA. The schema *ObjectCollection* introduces two variables describing HLA objects: *Objects*, the set of objects currently recognized by the RTI, and *ObjectsToClass*, the mapping that identifies the class associated with each object. The set *ObjectAttrs* contains all the object attributes related to the currently known objects, as required by the first state invariant. The two projection functions, *ObjectAttrToObject* and *ObjectAttrToClassAttr*, relate object attributes back to the corresponding objects and class attributes. As indicated by its double-headed arrow, *ObjectAttrToObject* is a function that maps onto its range: that is, every object has at least one object attribute, the one associated with the *privToDeleteObject* class attribute.

The final two state invariants define the required correspondence between object attributes, objects, and class attributes. The first of these constraints specifies that for any object and any class attribute defined by that object's class, a corresponding object attribute relates the object and the class attribute. The final constraint specifies that no two object attributes relate the same object and class attribute.

There is an alternative formulation of the object attribute construct. As shown in Figure 7.6, each object attribute could be viewed as an ordered pair, with the complete collection of object attributes constructed directly from the existing variables. However, NP does not support denoting a particular object attribute in this formulation, so I chose to use the otherwise more cumbersome representation shown in Figure 7.5.

$$\begin{array}{|l}
\mathit{ObjectAttrs} : \mathbb{P}\ (\mathit{OBJECT} \times \mathit{CLASSATTR}) \\
\hline
\mathit{ObjectAttrs} = \mathit{Objects} \lhd \mathit{ObjectToClass} \mathbin{;} \mathit{AttributesToClass}^{\sim}
\end{array}$$

Figure 7.6. Alternative model of object attributes

$[\mathit{FEDERATE}]$

$$\begin{array}{|l}
\hline
\textit{SimulationState} \\
\mathit{ObjectCollection} \\
\mathit{Federates} : \mathbb{P}\ \mathit{FEDERATE} \\
\mathit{Publishing}: \mathit{FEDERATE} \leftrightarrow \mathit{CLASSATTR} \\
\mathit{Owns} : \mathit{FEDERATE} \leftrightarrow \mathit{OBJECTATTR} \\
\hline
\end{array}$$

Figure 7.7. Model of (explicit) simulation state.

7.2.2 Required State

The simulation state, as shown in Figure 7.7, describes the state of the simulation that is explicitly described in the IFSpec. I chose to separate the explicit state from the implicit state (represented by the internal state given in Figure 7.8) for three reasons:

- Ease of validation. Because the simulation state is explicitly described in the IFSpec, it is easily checked against the informal specification (the IFSpec). The implicit state, on the other hand, requires significantly more effort to check against the original specification. By culling it out separately, the original specification writers are likely to pay closer attention to the implicit state.
- Isolation for analysis. Some claims require only the explicit state. By separating the implicit state, Ladybug can check claims by examining fewer cases.[3]
- Implementation freedom. The simulation state must be faithfully implemented in any actual code. Although the behavior of the implicit state is required in some form, the implementors have more freedom to choose an alternative structuring of this information in the final design.

The *SimulationState* schema introduces three new variables, but no new constraints. *Federates* is the set of federates currently joined in the simulation. *Publishing* and *Owns* describe the attributes published and owned by each federate, as described in the IFSpec. A full model would also introduce a variable describing the subscribe relations, but subscribing is irrelevant to the properties being checked and has been omitted.

Figure 7.8 lists the schema *OwnershipInternalState*, which models the implicitly described state. This schema includes two variables that indicate each federate's willingness to accept or divest ownership of a specific object attribute. These variables were introduced based on statements in the IFSpec such as

> The federate has informed the RTI of its intent to divest ownership of the specified attributes.

that appears in the post-condition of the description of the *Request Attribute Ownership Divestiture* service (see Figure 7.1). This state also records the set of federates that may gain ownership of an object attribute as indicated by the *Request Attribute Ownership Divestiture* service.

For convenience, I combine the explicit state and the implicit state into a single schema, *ExecutionState*.

[3] If Ladybug was able to remove irrelevent variables, this factor would be eliminated.

OwnershipInternalState
$WillingToDivest : FEDERATE \leftrightarrow OBJECTATTR$
$WillingToAccept : FEDERATE \leftrightarrow OBJECTATTR$
$TargetOwners : FEDERATE \leftrightarrow OBJECTATTR$

Figure 7.8. Model of (implicit) internal state.

ExecutionState
SimulationState
OwnershipInternalState

Figure 7.9. Model of complete required state.

7.2.3 Two Properties: NoTwoOwners and CompleteOwners

In the full model, I specify eight properties about the basic HLA system. In this section, I describe the two most significant of these properties in detail, *NoTwoOwners* and *CompleteOwners*. The remaining properties follow the same structure as the two presented here and can be seen in Appendix D.

I model all the properties as constraints on the state, including either the explicit state or the implicit state, or both. By describing the properties separately from the base description, I can pose questions about whether the system does or does not imply a specified property. Ladybug can also check whether operations (or sequences of operations) maintain these properties.

NoTwoOwners
SimulationState

$Owns\sim \in (OBJECTATTR \nrightarrow FEDERATE)$

Figure 7.10. Z model of unique ownership property.

The schema *NoTwoOwners*, given in Figure 7.10, is based on the IFSpec statement

> The privilege to update a value for an attribute is uniquely held by a single federate at any given time during a federation execution.

This property depends only on the explicit state that is described by the schema *SimulationState*. The condition on this property requires the inverse of *Owns* to be a function from *OBJECTATTR* to *FEDERATE*, implying that *Owns* itself is injective. This, in turn, implies that every object attribute is owned by no more than one federate. The designers of the HLA view this property as an invariant.

The second property detailed here, *CompleteOwners*, requires that every object attribute be owned by some federate. Figure 7.11 lists the model of *CompleteOwners*. Unlike most of the properties modeled, *CompleteOwners* is not required by the IFSpec and is not an invariant, as some services (including unconditional divestitures, unpublishing class attributes, and resigning from the federation) may leave object attributes unowned. However, this property is still worth considering as it should be invariant across some complete protocols, such as negotiated divestiture and acquisition. This property is checked by verifying that if every attribute is owned when a protocol begins (and there are no other concurrent services), then every attribute is owned when the protocol finishes.

$$\begin{array}{|l}
\hline
\mathit{CompleteOwners} \\
\mathit{SimulationState} \\
\hline
\mathrm{ran}\ \mathit{Owns} = \mathit{ObjectAttrs} \\
\hline
\end{array}$$

Figure 7.11. Z model of universal ownership property.

7.2.4 Two Operations: RequestAttrOwnDivestiture and AttrOwnDivestNotify

Of the ten total operations specified in the full model of ownership management, I detail two operations, *RequestAttrOwnDivestiture* and *AttrOwnDivestNotify*. These operations combine to implement the simplest unconditional divestiture protocol execution.

The description of these operations, as with all others, consists of three pieces: the arguments, the pre-conditions, and the post-conditions. I follow the Z convention, appending a question mark (?) to the end of the name of each input parameter. As detailed below, I choose to model only the subset of the parameters that is relevant to the analysis. I translate the pre-conditions into state invariants on the pre-state, again ignoring pre-conditions irrelevant to the needs. I similarly translate the post-conditions as state invariants of the post-state (as indicated by the primed variables).

$$\begin{array}{|l}
\hline
\mathit{RequestAttrOwnDivestiture} \\
\Delta\,\mathit{ExecutionState} \\
\mathit{fed?} : \mathit{FEDERATE} \\
\mathit{targets?} : \mathbb{P}\ \mathit{FEDERATE} \\
\mathit{obj?} : \mathit{Object} \\
\mathit{cattrs?} : \mathbb{P}\ \mathit{CLASSATTR} \\
\mathit{oattrs} : \mathbb{P}\ \mathit{OBJECTATTR} \\
\hline
\mathit{ObjectAttrToClassAttr}(\mathit{oattrs}) = \mathit{cattrs?} \\
\mathit{ObjectAttrToObject}(\mathit{oattrs}) = \{\mathit{obj?}\} \\
\mathit{fed?} \in \mathit{Federates} \\
\{\mathit{fed?}\} \times \mathit{oattrs} \subseteq \mathit{Owns} \\
\mathit{WillingToDivest}' = \mathit{WillingToDivest} \cup (\{\mathit{fed?}\} \times \mathit{oattrs}) \\
\mathit{WillingToAccept}' = \mathit{WillingToAccept} \\
\mathit{TargetOwners}' = \mathit{TargetOwners} \cup (\mathit{targets?} \times \mathit{oattrs}) \\
\mathit{SimulationState}' = \mathit{SimulationState} \\
\hline
\end{array}$$

Figure 7.12. The Z model of the *Request Attribute Ownership Divestiture* service.

The *Request Attribute Ownership Divestiture* service, which is described informally in Figure 7.1, allows a federate to notify the RTI that it (the federate) no longer wishes to be responsible for updating any of a set of object attributes. When the RTI responds with an invocation of the *Attribute Ownership Divestiture Notification*, the originating federate is no longer responsible for the object attributes in question.

Figure 7.12 shows the model of the *Request Attribute Ownership Divestiture* service. This operation requires all the execution state, including the implicit state described by *OwnershipInternalState*. The operation takes four inputs, *fed?*, the federate seeking to relinquish ownership, *targets?*, the set of potential new owners, *obj?*, the object whose attributes are being disowned, and *cattrs?*, a set of class attributes describing the object attributes to be disowned.

The IFSpec states that the *targets?* parameter is optional, but Z does not directly support optional parameters. To handle this complication, I assume that the set of all *Federates* is passed

$$
\begin{array}{|l}
\hline
\mathit{AttrOwnDivestNotify} \\
\Delta \mathit{ExecutionState} \\
\mathit{fed?} : \mathit{FEDERATE} \\
\mathit{obj?} : \mathit{Object} \\
\mathit{cattrs?} : \mathbb{P}\ \mathit{CLASSATTR} \\
\mathit{oattrs} : \mathbb{P}\ \mathit{OBJECTATTR} \\
\hline
\mathit{ObjectAttrToClassAtttr}(\mathit{oattrs}) = \mathit{cattrs?} \\
\mathit{ObjectAttrToObject}(\mathit{oattrs}) = \{\mathit{obj?}\} \\
\mathit{fed?} \in \mathit{Federates} \\
\{\mathit{fed?}\} \times \mathit{oattrs} \subseteq \mathit{Owns} \\
\\
\mathit{Owns}' = \mathit{Owns} \setminus (\ \{\mathit{fed?}\} \times \mathit{oattrs}) \\
\mathit{Objects}' = \mathit{Objects} \\
\mathit{Publishing}' = \mathit{Publishing} \\
\mathit{ObjectAttrs}' = \mathit{ObjectAttrs} \\
\mathit{Federates}' = \mathit{Federates} \\
\\
\mathit{WillingToAccept}' = \mathit{WillingToAccept} \\
\mathit{WillingToDivest}' = \mathit{WillingToDivest} \setminus (\ \{\mathit{fed?}\} \times \mathit{oattrs}) \\
\mathit{TargetOwners}' = \mathit{TargetOwners} \\
\hline
\end{array}
$$

Figure 7.13. The Z model of the *Attribute Ownership Divestiture Notify* service.

when no constraint is requested. I also choose to ignore two arguments specified in the IFSpec. The user-supplied tag, although important in any actual implementation, does not affect any interesting properties. The conditional divestiture flag is ignored here because it represents control flow, rather than the resulting structure. The model does not require divestiture (or disallow conditional divestiture), so the analysis will consider both cases of the flag.

The first four conditions capture the relevant pre-conditions, whereas the final four conditions capture the post-conditions. The first two conditions assert that the set of object attributes, referred to by *oattrs*, matches the object referred to by *obj?*, and the class attributes referred to by the set *cattrs?*. The third condition, $\mathit{fed?} \in \mathit{Federates}$, enforces the second pre-condition in the IFSpec

> The federate has joined the federation execution.

The fourth condition, $\{\mathit{fed?}\} \times \mathit{oattrs} \subseteq \mathit{Owns}$, enforces the IFSpec pre-condition

> The federate owns the specified attributes.

The next three conditions describe the change to the internal state. After the request, the federate is willing to divest the indicated object attributes, but there is no change in any federate's willingness to accept new ownership. This change, modeled by the fifth condition, is required by the IFSpec post-condition

> The federate has informed the RTI of its request to divest ownership of the specified attributes.

I assume that the willingness to accept ownership does not change due to this operation, as modeled by the next condition. This assumption, although reasonable and the likely intent of the specifiers, indicates that something is missing in the original specification; it is closely related to other assumptions described later in this section.

The target owners to consider, as described by the *targets?* parameter, are recorded, supporting option 1 in the IFSpec.

> The federate can specify which federate(s) can take ownership of the released attributes, otherwise any federate may own them.

```
[OBJECT, OBJECTATTR, FEDERATION]

┌─ObjectCollection ──────────────────────────────────────────
│ Objects : ℙ OBJECT
│ ObjectsToClass : OBJECT → CLASS
│ FederationObjects :  FEDERATION ⤔ OBJECT
│ ObjectAttrs : ℙ OBJECTATTR
│ ObjectAttrToObject: OBJECTATTR → OBJECT
│ ObjectAttrToClassAttr : OBJECTATTR → CLASSATTR
├───────────────
│ ObjectAttrs = dom (ObjectAttrToObject ▷ Objects)
│ ObjectToClass ;  AttributesToClass~ = ObjectAttrToObject~ ; ObjectAttrToClassAttr
│ ( ObjectAttrToObject ; ObjectAttrToObject~) ∩
│                      ( ObjectAttrToClassAttr ; ObjectAttrToClassAttr~)
│          ⊆ id OBJECTATTR
│ Objects = ran FederationObjects
└────────────────────────────────────────────────────────────
```

Figure 7.14. The *ObjectCollection* schema modified to consider multiple federations.

The final condition, *SimulationState' = SimulationState*, captures the IFSpec post-condition

> No change in attribute ownership.

Figure 7.13 shows the response service from the RTI. The *AttrOwnDivestNotify* operation takes four arguments, similar to those used as inputs to *ReqAttrOwnDivest* operation. The operation requires that the federate being notified is currently a member of the federation and owns the object attributes in question.

After the operation, the ownership has changed, with the target federate no longer owning the target object attributes. In addition, the *WillingToDivest* relation is updated, removing the pending desire to divest (which has now been fulfilled). This latter change is not specified in the IFSpec. In fact, version 1.2 of the IFSpec never states when a willingness to divest (or accept) should be cancelled (or maintained). Based at least partially on our feedback, version 1.3 of the IFSpec [DoD98] does state when these intentions should be cancelled. To progress in the analysis, I chose to specify "reasonable" points for cancelling the intentions.

In draft 4 of version 1.2 (no longer available), the IFSpec placed no pre-condition on the *Attribute Ownership Divestiture Notification* service requiring that the federate had previously attempted to divest ownership of the specified attributes. Draft 6 of version 1.2 partially repairs this flaw with the pre-condition

> A federate has previously attempted to divest ownership of the specified attributes.

This pre-condition is still flawed. It should require that the federate that currently owns the attribute has requested to divest ownership, without any subsequent cancellation of its willingness to divest.

It should not come as a surprise that both of these inconsistencies (as well as other, similar ones discovered with other services) arise in properties related to the state that has been only implicitly specified.

I discovered other ambiguities during this formalization. The IFSpec does not indicate if the RTI is allowed to satisfy the divestiture partially, leaving the original owner in posession of some of the attributes it was attempting to divest. It is similarly unclear if the RTI may combine multiple ownership divestiture requests, returning a single divestiture notification. I have assumed in our model that both possibilities are allowable, but some conforming implementations may disallow

$$
\begin{array}{|l}
\hline
\mathit{SimulationState} \\
\mathit{ObjectCollection} \\
\mathit{Federates} : \mathbb{P}\ \mathit{FEDERATE} \\
\mathit{Federations} : \mathit{FEDERATE} \rightarrow \mathit{FEDERATION} \\
\mathit{Publishing} : \mathit{FEDERATE} \leftrightarrow \mathit{CLASSATTR} \\
\mathit{Owns} : \mathit{FEDERATE} \leftrightarrow \mathit{OBJECTATTR} \\
\hline
\mathit{Federates} = \operatorname{dom} \mathit{Federations} \\
\hline
\end{array}
$$

Figure 7.15. The *SimulationState* schema modified to support multiple federations.

one or both variations. Modelling the most general possibility is a good guideline, but the more general model may yield spurious flaws.

7.2.5 Modeling Bridges

The two most notable changes to the model to support bridges is support for multiple federations and the model of the bridges themselves. Supporting multiple federations requires adding relations to map objects and federates to federations. A slight change in both *ObjectCollection* and *SimulationState* accommodates this difference. Figure 7.14 shows the final version of *ObjectCollection*, with one additional variable, *FederationObjects*, to map objects to the appropriate federation and two constraints on valid values of *FederationObjects*. Figure 7.15 shows the final version of *SimulationState*; the variable *Federations*, which maps federates to the federations, is new and constrains the set of valid federates.

The meat of the change comes with the introduction of the schema *BridgeState*. Figure 7.16 gives this new schema. The two key elements in *BridgeState* are bridges and maps. Each bridge represents the combination of the transformation manager and the FOMAT. The *SurrogateFor* relation associates each surrogate federate with a bridge. Each *MAP* represents a single distinct mapping in a FOMAT and generates one pair of objects in the relation *ObjectMapping*; the two objects are equiva-

$$[\,\mathit{BRIDGE}, \mathit{MAP}]$$

$$
\begin{array}{|l}
\hline
\mathit{BridgeState} \\
\mathit{SimulationState} \\
\mathit{Bridges} : \mathbb{P}\ \mathit{BRIDGE} \\
\mathit{SurrogateFor} : \mathit{FEDERATE} \twoheadrightarrow \mathit{BRIDGE} \\
\mathit{ObjectMapping} : \mathit{OBJECT} \leftrightarrow \mathit{OBJECT} \\
\mathit{MapsFromObject} : \mathit{MAP} \rightarrow \mathit{OBJECT} \\
\mathit{MapsToObject} : \mathit{MAP} \rightarrow \mathit{OBJECT} \\
\mathit{MapsForBridge} : \mathit{MAP} \rightarrow \mathit{BRIDGE} \\
\hline
\operatorname{ran} \mathit{SurrogateFor} = \mathit{Bridges} \\
\mathit{ObjectMapping} = \mathit{MapsFromObject}{\sim};\mathit{MapsToObject} \cup \mathit{MapsToObject}{\sim};\mathit{MapsFromObject} \\
\mathit{SurrogateFor};\mathit{SurrogateFor}{\sim} \cap \mathit{Federations};\mathit{Federations}{\sim} \subseteq \operatorname{id} \mathit{FEDERATE} \\
(\mathit{FederationObjects};\mathit{ObjectMapping};\mathit{FederationObjects}{\sim}) \cap \operatorname{id} \mathit{FEDERATION} = \varnothing \\
\mathit{MapsForBridge}{\sim};\ (\mathit{MapsFromObject} \cup \mathit{MapsToObject});\mathit{FederationObjects} \subseteq \mathit{SurrogateFor}{\sim};\mathit{Federations} \\
\hline
\end{array}
$$

Figure 7.16. The *BridgeState* schema describes the topology of bridges and the object mappings.

ObjectMappedOncePerFederation

BridgeState

$$\forall o : OBJECT \bullet FederationObjects \rhd ObjectMapping^{*}(o) \subseteq (FEDERATION \text{ ⇸ } OBJECT)$$

Figure 7.17. A desirable properties for bridged systems.

lent, but in different federations that are joined by the bridge. The three projection functions, *MapsFromObject*, *MapsToObject*, and *MapsForBridge*, describe the behavior of the map. Despite the *to* and *from* naming, the mappings implied are bidirectional, with *ObjectMapping* mapping *to* to *from* as well as *from* to *to*. I ignore many other aspects of the mapping, such as the mapping of attributes and classes across the bridge, but these mappings are one-to-one and should introduce no new issues.

The definition of a bridge in [SB+98] is very weak, so I have made assumptions about the restrictions that should be placed on bridges. The constraints require that every bridge has at least one surrogate. The constraint

$$SurrogateFor ; SurrogateFor^{\sim} \cap Federations ; Federations^{\sim} \subseteq \mathrm{id}\ FEDERATE$$

limits a bridge to at most one surrogate per federation. The first half of the left hand side of the inequality builds a relation mapping all surrogates to other surrogates supporting the same bridge. The relation on the right-hand side of the intersection relates any pair of federates that are in the same federation. The intersection therefore relates federates that are part of the same federation and serve the same bridge. By limiting this intersection to a subset of the identity function, every bridge is limited to having at most one surrogate in each federation.

The remaining constraints require that no mapping map two objects in the same federation and that the object mappings only map objects in federations for which the bridge has a surrogate.

The final step in supporting bridges is to define an interesting property that may or may not hold in bridged simulations. The property *ObjectMappedOncePerFederation* states that no object is mapped to more than one object (including itself) in any federation. Having the same effective object appear twice in a simulation could introduce obvious difficulties.

7.3 Analyzing the Formal Model

The final phase in this process is analyzing the model. This analysis consists of three steps, detailed in the following sub-sections: translating the Z model to Ladybug's input language NP, constructing claims about the model to be checked, and reviewing the results of checking the claims with Ladybug.

7.3.1 Translating the Z Model

In order to check the specification, Ladybug requires the specification to be written in the language NP. For many Z specifications, the translation from Z to NP is straightforward, mostly consisting of transliterating special Z symbols into the equivalent NP ASCII constructs. Only a few items within the translation are worth noting specifically. Appendix E gives the complete NP specification for the ownership management services. Appendix F gives the complete NP specification with bridges considered.[4]

[4] The specifications presented in the appendices include the change required to maintain membership in the federation realized during the analysis and vary slightly from the versions presented in this section.

Figure 7.18 shows the ObjectCollection schema, translated from the Z schema *ObjectCollection* given in Figure 7.5. NP does not support the axiomatic definitions used in Figure 7.4 (only schemas), so the initial definitions have been merged into the ObjectCollection schema. The *privToDeleteObject* attribute is never used by any of the properties considered, so I have not introduced any equivalent of the *privToDeleteObject* variable into the NP specification.

I separated two of the conditions in *ObjectCollection* into a separate schema, called GoodObjColl. As noted earlier, this separation enables these properties to be checked. Because all the later properties that I will check assume a sound ObjectCollection, the GoodObjColl schema, rather than the raw ObjectCollection schema, is imported into SimState. Figure 7.19 shows the NP translation of the explicit state into the schema SimState, as well as the implicit and total state.

To simplify the analysis, I model the operations as directly accepting a set of object attributes, rather than the actual set of class attributes. I feel that explicitly specifying the requirement to translate from class attributes to object attributes is important in the formal specification, as this translation could easily be missing or flawed in an actual implementation. However, a faulty translation from class attributes to object attributes is a flaw in the implementation, not the overall HLA design. This analysis is attempting to check the design, so this complication is unnecessary in the NP translation. Removing this complication both makes the NP specification easier to read and reduces the number of cases to be considered by Ladybug. Figure 7.20 shows the NP operation `RequestAttrOwnDivestiture`.

Because NP does not directly support cross products, some of the conditions within the opera-

```
/* Define the basic universe */
ObjectCollection = [
  Objects: set OBJECT
  Object_Attrs: set OATTR
  ObjectToClass: tot OBJECT -> CLASS
  ClassAttrsToClass: tot ATTR -> CLASS
  ObjAttrsToClassAttrs: tot OATTR -> ATTR
  ObjAttrsToObject: tot suj OATTR -> OBJECT
|
   /* Only object attributes about known objects are of interest */
   Object_Attrs = dom (ObjAttrsToObject :> Objects)
]


GoodObjColl = [
  ObjectCollection
|
  ObjectToClass;ClassAttrsToClass~ =
      ObjAttrsToObject~;ObjAttrsToClassAttrs
  ObjAttrsToObject;ObjAttrsToObject~ &
      ObjAttrsToClassAttrs;ObjAttrsToClassAttrs~ <= Id
/* First invariant says that each instance has the attributes
specified by its class (or has the right number of attributes).
second invariant states that the intersection of the
two equivalence relations on ObjAttrToObject and ObjAttrsToClassAttrs
intersect only when the same object attributes
are the subject, i.e., two object attributes can't be of the
same type and belong to the same object instance. */
]
```

Figure 7.18. The NP schemas ObjectCollection and GoodObjColl

```
/* Explicitly defined state */
SimState = [
  GoodObjColl
  Federates: set FED
  Publishing: FED <-> ATTR
  Owns: FED <-> OATTR
]
/* Implicitly defined state */
OwnershipInternalState = [
  WillingToDivest:FED <-> OATTR
  WillingToAccept: FED <-> OATTR
  TargetOwners : FED <-> OATTR
]
/* Total state to consider */
ExecutionState =
[SimState
OwnershipInternalState]
```

Figure 7.19. The NP specification of the explicit, implicit, and total state.

```
RequestAttrOwnDivestiture(fed?:FED, obj?:OBJECT, targets?:set FED,
                                                        oattrs?:set OATTR) =
[
  ExecutionState
  const SimState
|
  fed? in Federates
  ObjAttrsToObject.oattrs? = {obj?}
  /* ({fed?} <: Un :> oattrs?) is the same as Z {fed?} x oattrs */
  ({fed?} <: Un :> oattrs?) <= Owns
  WillingToDivest' = WillingToDivest U ({fed?} <: Un :> oattrs?)
  WillingToAccept' = WillingToAccept
  TargetOwners' = TargetOwners U (targets? <: Un :> oattrs?)
]
```

Figure 7.20. The NP specification of the *Request Attribute Ownership Divestiture* service.

tion definitions must be recast slightly. The Z expression

$$\{fed?\} \times oattrs$$

can be translated to the NP expression

```
{fed?} <: Un :> oattrs?
```

where Un is the universal relation, forced by context to be typed as FEDERATE <-> OATTR.

The bridge extensions introduce another issue in the translation. The property *ObjectMappedOncePerFederation* uses a universal quantifier to require that each object is mapped only once per federation. This Z constraint is expressed as

$$\forall o : OBJECT \bullet FederationObjects \rhd ObjectMapping^{*}(o) \subseteq (FEDERATION \nrightarrow OBJECT)$$

NP offers no support for explicit quantifiers. Although no translation approach can convert an arbitrary quantified formula to an equivalent quantifier-free formula, some formula can be converted. For this property, the NP formula

```
/* Allow bridges between federations */
BridgeState =
[
 SimState
 Bridges : set BRIDGE
 SurrogateFor : FED -> BRIDGE
 ObjectMapsFrom : MAP -> OBJECT
 ObjectMapsTo : MAP -> OBJECT
 ObjectMapping : OBJECT <-> OBJECT
 BridgeMapping : tot MAP -> BRIDGE
|
 ran SurrogateFor = Bridges
 ObjectMapping = ObjectMapsFrom~ ; ObjectMapsTo

  /* A bridge has one surrogate for each federation it participates in */
 inj SurrogateFor~;Federations

 /* Each object mapping must be across different  federations */
 (FederationObjects;ObjectMapping;FederationObjects~) & Id = {}

  /* Each federation represented must correspond to a surrogate federate */
 BridgeMapping~;ObjectMapsFrom;FederationObjects~ <= SurrogateFor~;Federations
 BridgeMapping~;ObjectMapsTo;FederationObjects~ <= SurrogateFor~;Federations
]
```

Figure 7.21. The NP BridgeState schema, which was adjusted to remove the set of triples.

```
(FederationObjects~ ; ( ObjectMapping+\Id) ; FederationObjects) & Id = {}
```

imposes the same constraint. The closure of ObjectMapping is a relation that maps all objects to all equivalent objects (as mapped by the FOMAT). By removing the identity from this relation, each object is mapped to every equivalent object except itself. Bracketing this relation with the FederationObject relation extends it to federations. If this composite relation maps any federation to itself, an object must be equivalent to another object in the same federation.

7.3.2 Constructing the Claims

I constructed two kinds of claims about the ownership management specification. The simpler claims assert that a property is invariant across any possible invocation on a single operation. The more complicated claims describe an entire protocol execution, asserting that some property holds after the entire execution if the property holds prior to the execution.

Figure 7.22 shows a simple operation invariant claim written in NP. Unlike schemas in NP, which use an equals sign (=) to separate the header of the schema from its body, claims in NP separate the header from the body with a double colon (::). The AttrDivNotSoundOwns claim asserts that the properties described in the schema SoundOwners (which requires unique valid ownership, but not universal ownership) is invariant across the *Attribute Ownership Divestiture Notify* service (as described in the NP schema AttrOwnDivestNotify). The claim must hold for any federate, object, or set of object attributes, using the federate, object, and set of object attributes as the arguments to AttrOwnDivestNotify.

The remaining claims are more complex, as they check properties that are not invariants, but should hold at specified points during the protocol. I do not recheck properties that have been shown invariant across all operations, as they must also hold invariant across any combination of operations that comprise a complete protocol. Figure 7.23 shows the NP claim that asserts that a conditional divestiture protocol does not lose ownership of objects.

The five indented lines near the end of the claim describe one possible sequence of services for

```
/* Check for complete ownership after a simple conditional divestiture */
ConditionalCompleteOwners(fed1:FED, fed2:FED, targets : set FED,
                          obj:OBJECT, oattrs1:set OATTR,
                          oattrs2:set OATTR)::
[
 ExecutionState
|
 /* require the case we are interested in */
 not fed2 = fed1 and
 fed2 in targets and
 oattrs2 <= oattrs1 and
 SoundOwners and
 CompleteOwners and
 /* conditional divestiture of oattrs1, actually divesting oattrs2 */
   (RequestAttrOwnDivestiture(fed1,obj,targets,oattrs1);
    RequestAttrOwnAssumption(fed2,obj,oattrs1);
    RequestAttrOwnAcquisition(fed2,obj,oattrs2);
    AttrOwnDivestNotify(fed1,obj,oattrs2);
    AttrOwnAcquisitionNotify(fed2,obj,oattrs2)) =>
 CompleteOwners'
]
```

Figure 7.23. The NP claim asserting that all object attributes are owned after a conditional divestiture.

this protocol. A federate (fed1) initiates the protocol by requesting conditional divestiture of a set of object attributes (oattrs1). The RTI then requests that a second federate (fed2) assume ownership of those attributes. The second federate agrees to take ownership of a subset of the attributes (oattrs2) and requests that ownership from the RTI. The RTI can then respond to the first federate with a divestiture notification of the subset of attributes. Finally, the RTI grants ownership of a subset of the object attributes divested to the second federate.

Because the bridge extensions do not change the definitions of the existing services, any property guaranteed by the original definition will hold in the bridge-extended definitions as well.

```
/* Check if the non-empty state allows two owners */
NoTwoOwners = [NonEmpty | fun Owns~]
NoBadOwnedAttrs  =  [SimState | ran Owns = Object_Attrs]
NoBadOwners = [SimState | dom Owns <= Federates]
OwnsOnlyIfPublishes = [SimState | Owns;ObjAttrsToClassAttrs <= Publishing ]
SoundOwners = [
 NoTwoOwners
 NoBadOwnedAttrs
 NoBadOwners
 OwnsOnlyIfPublishes
]
AttrDivNotSoundOwns(fed:FED, obj:OBJECT, oattrs:set OATTR)::
  SoundOwners and AttrOwnDivestNotify(fed,obj,oattrs) => SoundOwners'
```

Figure 7.22. The NP sound ownership properties and the NP claim that the sound ownership properties are invariant across the *Attribute Ownership Divestiture Notify* service.

After rechecking the previous claims, I added a new claim to the bridge specification. Figure 7.24 shows the property ObjectMappedOncePerFederation and the related claim that checks that the definition of a bridged simulation satisfies this property. This claim is different from those expressed previously; it tests the definition of bridged simulations, rather than the behavior of any operation. Unlike previous claims, once verified, this claim will hold for any future operations introduced as well. However, a counterexample to this claim discovered by Ladybug may not be realizable; no sequence of operations introducing the erroneous state is demonstrated.

7.3.3 The Analysis

Analyzing the claims with Ladybug is nearly automatic. The only significant choice left to the analyst at this stage is to bound the number of elements to be considered by Ladybug in the analysis.

The default scope assumes three elements of each type, which suffices for many specifications. For the ownership management specification, however, I chose to vary the scope from the default for three reasons:

1) to satisfy the requirements of the specification

2) to gain more confidence in the analysis and

3) to reduce the time required by the search.

For these claims, I limit the number of classes to one, as ownership of attributes is independent of class and therefore class distinctions are irrelevant to the properties being checked. I limit the number of federates and class attributes to two apiece, the minimum number that allows divided ownership of a single object. I limit the number of objects to three. These restrictions in turn require the support of six object attributes (three objects with two object attributes apiece). When choosing the scope, interactions such as this must be carefully noted if the analysis is to be trusted. Limiting the number of object attributes to fewer than six would force the analysis to consider fewer than three objects or two class attributes in order to satisfy the other requirements.

For the bridge claims, the number of object attributes is irrelevant. However, enough bridges, federates, maps, and objects are required to allow the claims to be violated. Starting with three federations, I require seven federates to allow two surrogates in each federation plus one federate that is not a surrogate. Three bridges allow many interesting topologies to be built for these federations. Note that not all the federations need to be considered in every case. Six objects allow two objects in each federation, allowing the same object to effectively appear twice. Six maps allows each bridge to map each pair of objects in the federations bridged.

Ladybug completed the analysis quickly using both the default and selected scopes. Most of the checks required a few seconds to complete (when run on a 400Mhz Pentium II using the Sun JDK 1.1.8).

The first issue arising from the analysis involves the set of federates joined in the federation.

```
/* The bridge property */
ObjectMappedOncePerFederation =
[
 BridgeState
|
 (FederationObjects~ ; (ObjectMapping+\Id) ; FederationObjects) & Id = {}
 ]
CheckObjectMapping:: BridgeState => ObjectMappedOncePerFederation
```

Figure 7.24. The claim about the bridge state specification.

```
Found Counterexample to Claim
AttrAcqNotSoundOwns:
ClassAttrsToClass : tot ATTR->CLASS =
            { a0 -> c0,
             a1 -> c0 }
ClassAttrsToClass' : tot ATTR->CLASS =
            { a0 -> c0,
             a1 -> c0 }
fed : FED =
            f0
Federates : set FED =
            { f0 }
Federates' : set FED =
            { f0 }
oattrs : set OATTR =
            { oa0 }
obj : OBJECT =
            ob0
ObjAttrsToClassAttrs : tot OATTR->ATTR =
            { oa0 -> a0,
             oa1 -> a1,
             oa2 -> a0,
             oa3 -> a1,
             oa4 -> a0,
             oa5 -> a1 }
ObjAttrsToClassAttrs' : tot OATTR->ATTR =
            { oa0 -> a0,
             oa1 -> a1,
             oa2 -> a0,
             oa3 -> a1,
             oa4 -> a0,

             oa5 -> a1 }
ObjAttrsToObject : tot OATTR->OBJECT =
            { oa0 -> ob0,
             oa1 -> ob0,
             oa2 -> ob1,
             oa3 -> ob1,
             oa4 -> ob2,
             oa5 -> ob2 }
ObjAttrsToObject' : tot OATTR->OBJECT =
            { oa0 -> ob0,
             oa1 -> ob0,
             oa2 -> ob1,
             oa3 -> ob1,
             oa4 -> ob2,
             oa5 -> ob2 }

Object_Attrs : set OATTR =
            { oa0, oa1 }
Object_Attrs' : set OATTR =
            { oa0, oa1 }
Objects : set OBJECT =
            { ob0 }
Objects' : set OBJECT =
            { ob0 }
ObjectToClass : tot OBJECT->CLASS =
            { ob0 -> c0,
             ob1 -> c0,
             ob2 -> c0 }
ObjectToClass' : tot OBJECT->CLASS =
            { ob0 -> c0,
             ob1 -> c0,
             ob2 -> c0 }
Owns : FED<->OATTR =
            { }
Owns' : FED<->OATTR =
            { f0 -> {oa0 } }
Publishing : FED<->ATTR =
            { }
Publishing' : FED<->ATTR =
            { }
TargetOwners : FED<->OATTR =
            { f0 -> {oa0 } }
TargetOwners' : FED<->OATTR =
            { }
WillingToAccept : FED<->OATTR =
            { f0 -> {oa0 } }
WillingToAccept' : FED<->OATTR =
            { }
WillingToDivest : FED<->OATTR =
            { }
WillingToDivest' : FED<->OATTR =
            { }
```

Figure 7.25. Output demonstrating a counterexample to the claim AttrAcqNotSoundOwn discovered by Ladybug.

The IFSpec never explicitly states that this set is unchanged by the execution of these operations, although the clear intent is that the only means that a federate can leave the federation is through the use of the *Resign Federation Execution* service. I adjusted the NP specification to capture this requirement.

With that omission repaired, I again analyzed the specification. As indicated by the output shown in Figure 7.25, the Ladybug analysis found that the *Attribute Ownership Acquisition Notify* service does not maintain the sound ownership properties invariant. In particular, the RTI may grant ownership of an object attribute to a federate that is not publishing the corresponding class attribute. This case can most clearly be seen by noticing that the Publishing' relationship is empty, indicating that no class attributes are being published and therefore no object attributes can be owned.

Reviewing the IFSpec, the only relevant pre-condition for this service is

> The federate has previously attempted to acquire ownership of the attribute.

Publishing the corresponding class attribute is a pre-condition of requesting ownership, so this combination might be expected to hold the property invariant for any actual protocol execution. However, if the federate unpublishes the class after requesting ownership, but prior to being granted ownership, a condition occurs where a federate can be granted ownership of an object attribute while not publishing the corresponding class attribute. An instance of this problem can be shown with the counterexample generated for the UnpublishInAcquisition claim.

In repsonse to this analysis, version 1.3 of the IFSpec adds the pre-condition

> The federate is publishing the corresponding class attributes at the known class of the specified object instance.

An alternative solution to this inconsistency is to have the unpublish operation cancel the federate's willingness to acquire any related object attributes.

Table 7.1 summarizes the results of the Ladybug runs. The columns listing the names of given types indicate the scope limit given for those types. The times given are in seconds. The entire state space was checked for all claims except for the case presenting a counterexample, where the search was halted after the first counterexample was found. Except for the indicated scopes, all checks were made using the default Ladybug settings.

The analysis of the bridge claim is more problematic. After several minutes of analysis, the chosen scope is clearly too large for simple analysis by Ladybug. Reconsidering the scope, I chose a smaller scope that offers less assurance of finding bugs, but one that should allow Ladybug to exhaust the space quickly. A quick search allows simple issues to be resolved quickly. The smaller scope considers bridging only two federations. Following the same logic as previously yields the scope

```
#CLASS=1 #ATTR=1 #FED=4 #FEDERATION=2 #BRIDGE=2 #MAP=3 #OBJECT=3
#OATTR=3
```

Ladybug now discovers a bug in a few seconds. If a cycle exists in the bridges, an object can be propagated around the cycle and be reintroduced as a duplicate in the original federation. One question remains: if no cycles in the bridges exist, can an object appear in a federation multiple times?

Figure 7.26 lists the property that the bridges are acyclic and a new claim that checks this question. The smaller scope now discovers no counterexamples. Returning to the larger scope, the analysis requires 21 hours and discovers no counterexamples. Therefore, requiring no cycles in the bridge configuration eliminates this potential problem.

This analysis has not checked many interesting potential problems of bridging federations.

Claim	Description	ATTR	CLASS	FED	OATTR	OBJ	Time (secs)
ReqAttrDivSoundOwns	Check that the *Request Attribute Divestiture* service maintains sound ownership.	2	1	2	6	3	7.1
ReqAttrAcqSoundOwns	Check that the *Request Attribute Acquisition* service maintains sound ownership.	2	1	2	6	3	19.8
AttrDivNotSoundOwns	Check that the *Attribute Divestiture Notification* service maintains sound ownership.	2	1	2	6	3	3.2
AttrAcqNotSoundOwns	Check that the *Attribute Acquisition Notification* service maintains sound ownership.	2	1	2	6	3	1.2*
PublishSoundOwns	Check that the *Publish Object Class* service maintains sound ownership.	2	1	2	6	3	2.6
UnpublishSoundOwns	Check that the *Unpublish Object Class* service maintains sound ownership.	2	1	2	6	3	2.2
ConditionalCompleteOwners	Check that a simple conditional divestiture protocol execution maintains complete ownership.	3	1	2	3	1	0.1
UnconditionalSoundTargets	Check that a simple unconditional divestiture protocol execution leaves all targets unowned.	3	1	2	3	1	0.1
ConditionalSoundTargets	Check that a simple conditional divestiture protocol execution leaves the targets set sound.	3	1	2	3	1	4.4
CheckObjectMapping	Check that no two objects in a federation are equivalent. #FEDERATION=2 #BRIDGE=2	1	1	4	3	3	5.1
CheckAcyclicObjMaps	Check that no two objects in a federation are equivalent if no cycles are allowed in the bridges. #FEDERATION=2 #BRIDGE=2	1	1	4	3	3	10.1
CheckAcyclicObjMaps	Check that no two objects in a federation are equivalent if no cycles are allowed in the bridges. #FEDERATION=3 #BRIDGE=3	1	1	7	6	6	21 hours

Table 7.1: Summary of the checks done by Ladybug. The time for AttrAcqNotSoundOwns and CheckObjectMapping is the time required to find the first counterexample.

Most notable among these potential problems are possible race conditions. The relational formula language used by Ladybug cannot express concurrency, so Ladybug is an inappropriate tool to check these issues. Instead, these checks should be made with a tool such as FDR [FDR97], a model checker for CSP specifications.

```
/* Check only acyclic bridge configurations */
NoBridgeCycles =
[
  BridgeState
 |
  (((SurrogateFor;SurrogateFor~)\Id);((Federations;Federations~)\Id))+ & Id = {}
]
CheckAcyclicObjMaps:: NoBridgeCycles => ObjectMappedOncePerFederation
```

Figure 7.26. An additional property and claim checking only acyclic bridge configurations.

7.4 Conclusions

I discovered inconsistencies and ambiguities during the formalization and analysis of an informal specification. While generally minor in nature, these flaws could introduce significant difficulties into the HLA development, if not caught at design time. Components, being developed by disparate organizations with possibly disparate interpretations of the IFSpec, could fail when joined together, forcing expensive testing and re-writes. With help from our feedback, these issues have been resolved in a new version of the IFSpec [DoD98].

Not surprisingly, most of the issues uncovered involved the implicitly-specified state. In my experience, a lack of detailed consideration of portions of the system leads to many of the flaws in system designs. Informal specifications facilitate this lack of consideration by allowing seemingly obvious portions of the system to remain unspecified or implicitly specified. Formalization helps identify these missing pieces.

Ladybug discovered two counterexamples to properties that were expected to be valid. The first counterexample demonstrated a flaw in the preconditions for the *Attribute Ownership Acquisition Notification* service, where a federate may have stopped publishing the underlying attribute before acquiring ownership of the object attribute. The second counterexample demonstrated a difficulty with the bridge concept if a cycle is allowed in the bridge topology.

In addition to these counterexamples, Ladybug made two notable contributions to the analysis:

- As a forcing function. Without the forced rigor of formalization and checking, it is far easier to ignore subtleties. In this example, the problems with mismatch in the set of object attributes became apparent when attempting to define specific protocol executions.
- Increased assurance of the correctness of the design. A lack of flaws is the eventual goal of any design. Assurance that at least selected possible flaws are not present is a significant first step. Experiences with analysis of other specifications, such as the new Mobile IPv6 standard [JNW98], have shown that flaws in systems can be discovered using automated checkers.

However, any automated analysis tool has fundamental limitations that should also be kept in mind:

- Only properties explicitly described are checked. Many flaws that have not been considered may remain. Ladybug cannot *generate* interesting claims, but rather can only *check* claims made by the analyst.
- The structure of the specification may hide flaws allowed by the design. As an example, the structure of the Z model requires that the services be treated atomically, with no concurrent interaction. With a distributed system such as HLA, such interactions are likely, leaving possi-

ble flaws undetected. Performing analyses of the same system with multiple tools and formalisms can help reduce these holes.

- The specification itself may be consistent, but it may not correctly capture the intent of the designers. Actual implementations, developed by humans who may understand that intent, may introduce flaws present in the design, but not captured in the formal specification.
- Finite checkers, such as Ladybug, place bounds on the problem to enable analysis. Flaws may exist only in systems that exceed those artificial bounds. Although I have yet to find a flaw in a design missed by a reasonable scope in any analysis, such flaws certainly exist in at least some designs.

In summary, the formalization and subsequent analysis discovered some flaws in the IFSpec and achieved a reasonable level of assuredness that other potential flaws are not present.

Additional analyses of these specifications are possible. Many other protocol sequences are meaningful and could be checked. Manual generation of these protocol executions is tedious and time-consuming. Due to time constraints, I chose to investigate only a handful of these possible protocol executions at this time. An ideal analysis tool could consider both a CSP specification, which can express the possible protocol executions succinctly, and a Z specification, which can express the outcomes of a particular protocol execution succinctly, and thus automate this task. One possible future path is the generation of interesting executions using the CSP checker FDR [FDR97], with a manual, or possibly automated, conversion of the output into NP claims.

Chapter 8

Conclusions

This chapter concludes the dissertation. Section 8.1 summarizes the contributions of this thesis. Section 8.2 describes how future work could enhance Ladybug by extending the input language or improving the search techniques. Section 8.3 briefly describes the application of selective enumeration to other problem domains.

8.1 Contributions

The thesis that I have presented in this dissertation is that selective enumeration makes some otherwise intractable searches feasible. Using only short circuiting and derived variables, most of the searches required by the benchmark suite are intractable. Bounded generation significantly reduces the cost of the tree pruning accomplished by short circuiting. No previous symmetry reduction techniques yielded even a small fraction of the time savings gained by the isomorph elimination used in Ladybug. With this level of reduction, Ladybug has been used to find bugs in real systems.

Beyond validating the thesis, this research offers four significant contributions: the approach, the algorithms, the tool, and the benchmark suite. Each contribution extends the platform for further research in a different direction.

The selective enumeration approach provides a framework for thinking about reducing the search space for a range of problems. Explicitly considering duplications and how they may be exploited to reduce the search space quickly led me to find effective ways of reducing the search space for the planning problem, discussed in Section 8.3.

The realization of selective enumeration for the relational domain led to several new algorithms. Three algorithms in particular offer significant advances over previous related approaches: bounded generation, domain coloring, and the isomorph-eliminating function generator. Bounded generation significantly reduces the number of values generated when compared to previous tree pruning approaches such as short circuiting and supports a much larger value space than is feasible with traditional constraint propagation. Both the domain coloring and the isomorph-eliminating generator offer sound and nearly perfect approximations to isomorph-free generation at a significantly cheaper cost than provided by any exact algorithms.

The Ladybug tool provides a realization of these algorithms that can solve interesting problems. Ladybug has been released to a handful of researchers thus far and will be made broadly available shortly. Ladybug, and its predecessor Nitpick, have been used in courses in the Masters of Software Engineering program at Carnegie Mellon University for five years.

The benchmark suite gives a broad blanket for comparing different techniques for analyzing relational specifications. The opportunity to add additional solvers to Ladybug makes an apples-to-apples comparison simpler.

In summary, this research provides both a tangible tool for aiding practitioners and students and a platform upon which further research can be based. This research could include further attempts to analyze relational specifications or work on reducing the search space for other purposes.

8.2 Improving Ladybug

This section describes how further work could improve Ladybug. Extensions to the input language NP would allow some specifications to be expressed more simply. Further reductions in the search space would allow more specifications or larger scopes to be checked. Each approach extends the applicability of Ladybug and simplifies its usage.

Two desirable extensions to the formula language became obvious in the HLA case study: quantifiers and n-ary functions and relations. Replacing these constructs in the translation from Z to NP was the most challenging effort in using Ladybug for the case study.

Quantifiers fit naturally into the framework established in this dissertation and could be implemented easily in Ladybug. The current search solves the existential quantifiers implicit in the negated formula. Adding explicit existential quantifiers is straightforward. Adding support for universal quantifiers is only slightly more difficult (and expensive). Isomorph elimination remains unchanged; a predicate found to be universally true for an isomorphically reduced set of assignments will also be universally true for the full set of assignments. Partial-assignment techniques can prune subtrees for which the predicate is guaranteed to be false at least once, instead of only pruning subtrees where all subtrees fail to satisfy the predicate of an existential quantifier. In the case of searches that discover counterexamples, the savings of this greater pruning will be at least partially offset by the need to fully expand the subtree for universally quantified variables.

Support for n-ary relations requires no change to the overall framework or to the partial-assignment techniques, although adding new bounded generation rules would be beneficial. The focus of the change would be new generators, including new isomorph-eliminating generators. The approach described in Chapter 6 extends to handle n-ary functions in a straightforward manner as curried functions. As an example, this approach begins the generation of two argument functions by choosing all possible domain sets for the first argument. Each element in each first argument domain set is mapped to all isomorphically distinct functions mapping the second argument to the result.

Two other extensions to NP would allow Ladybug to analyze specifications not considered in this dissertation: hierarchical given types and support for natural numbers. Allowing a hierarchy of given types simplifies modeling object-oriented designs. Support for this hierarchy is straightforward; non-root given types become distinct subsets of the corresponding root given type. Wide-scale usage of these non-root given types would reduce the efficiency of Ladybug, especially for the data structures representing relations and for isomorph elimination. Recovering some efficiency would be an engineering challenge in supporting this feature.

Natural numbers represent the largest challenge of any of the potential enhancements to NP. Any finite set of natural numbers is not closed under addition; ensuring that this issue does not introduce unsoundness into the search is a significant issue that would need to be addressed be addressed in the framework. How to effectively exploit the natural numbers in bounded generation is an implementation challenge. Although no pure symmetry exists in the natural numbers, ranges of numbers may be indistinguishable for a variable for any given formula. Exploiting this limited symmetry in isomorph elimination is another open research question.

Exploiting similar formula-specific symmetries is one possible approach to improving the reduction gained in Ladybug. Jackson considered term symmetries in [JJD98], but only found a small additional reduction. Further research is required to see whether this or other formula-specific symmetries can yield larger savings.

The experiences gained in this research have opened other questions about improving the search techniques. One question was raised in Chapter 6: is there an efficient variable ordering heuristic that exhibits fewer anomalies? Further investigations are needed to evaluate both the frequency and extent of the increase in the search space yielded by the current ordering heuristic as opposed to the search space yielded by an optimal ordering.

Chapter 6 also raised an issue with the ineffectiveness of bounded generation at exploiting some classes of formula, notably cyclic relations and compositions of relations. Cyclic (or acyclic) relations are a common requirement; adding special support for these cases is well justified. One possible approach is the construction of specialized generators that generate only cyclic or acyclic relations.

Chapter 5 briefly explored the promise and problems of multiple-antecedent rules for consequence closure. If a sufficiently efficient mechanism can be implemented that utilizes these rules, the partial-assignment reductions would be significantly improved for some searches.

8.3 Other Problem Domains

Search is one of the most common problems in computer science. Other search problems are also good candidates for solving with selective enumeration. In some cases, the existing selective-enumeration framework could be used intact; new techniques need to be developed to exploit the characteristics of the formula language that describe the new problem domain. Other cases would require extending the selective enumeration framework in some way. This section briefly describes how selective enumeration could solve three different search problems: test suite generation, shape analysis, and planning problems.

If an application or framework has been fully specified in a formal language such as Z, an opportunity exists to automatically generate a test suite to partially validate the application. As with any form of testing, complete validation is an unrealistic goal; testing every possible set of inputs is intractable for any interesting application. Instead, partition testing splits the universe of inputs into equivalence classes; the testing assumes that the application will behave correctly for every input in a equivalence class or will exhibit a bug for every input in the equivalence class.

Black [ABW98] proposed a variant of mutation testing that defines a partitioning of the inputs based on a formal specification of the application. Specification mutation testing, like source code mutation testing, assumes that the specification is almost correct. Any errors in the application are assumed to implement a mutation of the target specification that represents only a small change from the actual specification. For each possible mutation of the specification, the test suite contains a test that distinguishes the mutation from the desired specification. If the original specification is described by a formula spec and the mutation is described by a formula called mutant, each test is a solution to a formula defined as mutant and not spec.

Selective enumeration can help generate a test suite in two ways. As used in Ladybug, selective enumeration can find satisfying assignments to the formulae that describe the distinction between the desired and mutated specifications. Selective enumeration can also be used to reduce the size of the test suite, a traditional problem with generated test suites. Many inputs can distinguish the desired specification from many potential mutants. Inputs that test mutants already tested by other inputs are duplicates; by introducing a richer sense of duplication, selective enumeration can reduce the total size of the test suite. Only further research can determine whether selective enumeration can reduce the size of the test suite significantly.

Shape analysis is a second existing problem area that can be solved by selective enumeration. Sagiv [SRW99] describes how a program can be reduced to a series of formulae that describe how the program manipulates a single data structure. Any solution to these formulae describes the shape of the data structure. The formula language used by Sagiv is similar to the relational formula language used in this dissertation. Jackson and Vaziri [JV00] explore a similar approach using the boolean satisfiability solver in ALCOA. Selective enumeration can solve these formulae to discover the shape of data structures. For interesting programs, these formulae are probably very large. This size may require further enhancements to selective enumeration to support a tractable analysis of the larger problems.

The third problem domain that I have considered is the traditional STRIPS planning world [FN71]. In this domain, the world is described as a group of objects that have attributes such as position and operations such as move. Each operation makes some well defined changes to the object attributes. The initial state and goal state are described as constraints on the object attributes. The planning problem is to find a sequence of operations, called a plan, that will change the initial state to the goal state.

Finding a plan of n steps is equivalent to finding a solution to a formula with n variables, where each variable indicates the operation or operations performed at the corresponding time step. Again, selective enumeration can be applied to this formula. Using an iterative deepening approach, a selective enumeration tool can find a minimal length plan that solves the problem.

Because the formula language is very different from the formula language used in this thesis, selective enumeration would need to exploit different duplications than those exploited in Ladybug. Although the isomorph duplication would be applicable, this problem domain also exhibits partial-order symmetries between time steps. Therefore, two plans that vary only in the order of two operations that do not interact are duplicates. Some operations are non-reversible or can be reversed only with some minimum number of time steps. Duplications that exploit these requirements become the equivalent of the partial-assignment duplications described in Chapter 3.

Although the actual performance of a planner based on selective enumeration remains to be shown, I have performed rough experiments to show that the performance on some existing problems would be at least competitive with graph-plan-based tools [BF97], the current leading approach. In addition, selective-enumeration-based planners would not exhibit some of the restrictions on dynamic problems imposed by graph-plan-based planners. Success of a straightforward application of selective enumeration to a very different problem domain such as planning would demonstrate the power of duplications as a model for thinking about search reduction.

Appendix A: Collected Definitions

***Definition 2.1** (*U - for Relational Problem Domain*)

The universe of atomic values, *U*, is a finite set of unstructured and uninterpretted elements.

Definition 2.2** (Value - for Relational Problem Domain***)

$Value_{\text{scalar}} = U$

$Value_{\text{set}} = \mathbb{P}\, U$

$Value_{\text{rel}} = \mathbb{P}\, (U \times U)$

$Value = Value_{\text{scalar}} \cup Value_{\text{set}} \cup Value_{\text{rel}}$

Definition 2.3 (***Variable***)

The set *Variable* includes all variables used in the formula being solved.

Definition 2.4 (***N***)

$N = \mid Variable \mid$

Definition 2.5 (***Typing***)

The function $Typing : Variable \longrightarrow \mathbb{P}\ Value$ describes the subset of values that may be bound to each variable.

Definition 2.6** (Variable - for Relation Domains***)

Var_{scalar}, Var_{set}, and Var_{rel} partition the set *Variable*, with the constraints that

$$\forall \text{v} \in Var_{\text{scalar}} \,.\, Typing\,(\text{v}) \subseteq Var_{\text{scalar}}$$
$$\forall \text{v} \in Var_{\text{set}} \,.\, Typing\,(\text{v}) \subseteq Var_{\text{set}}$$
$$\forall \text{v} \in Var_{\text{rel}}.\ Typing\,(\text{v}) \subseteq Var_{\text{rel}} Value_{\text{scalar}})$$

Definition 2.7 (***A***)

The set of assignments $A : \mathbb{P}(Variable \twoheadrightarrow Value)$ is defined as

$$\{\ \text{a} : Variable \twoheadrightarrow Value \mid \forall(\text{var},\text{value}) \in \text{a}.\ \text{value} \in Typing(\text{var})\ \}$$

Definition 2.8** (Term***)

$Term = Term_{\text{scalar}} \cup Term_{\text{set}} \cup Term_{\text{rel}}$

$$\begin{aligned} Term_{\text{scalar}} \quad &::= \quad Var_{\text{scalar}} \\ Term_{\text{set}} \quad &::= \quad Var_{\text{set}} \,\Big|\, \{\ Term_{\text{scalar}}\ \} \,\Big|\, \{\,\} \,\Big|\, \text{Un} \,\Big| \\ &\qquad (\ Term_{\text{set}} \cup Term_{\text{set}}\ ) \,\Big|\, (\ Term_{\text{set}}\ \&\ Term_{\text{set}}\ ) \,\Big| \\ &\qquad (\ Term_{\text{set}} \setminus Term_{\text{set}}\ ) \,\Big|\, Term_{\text{rel}}(Term_{\text{set}}) \,\Big| \end{aligned}$$

dom $Term_{rel}$ | ran $Term_{rel}$

$Term_{rel}$ ::= Var_{rel} | ($Term_{set}$ <: $Term_{rel}$) | Id |
($Term_{rel}$ U $Term_{rel}$) | ($Term_{rel}$ & $Term_{rel}$) |
($Term_{rel} \backslash Term_{rel}$) | ($Term_{rel}$; $Term_{rel}$) |
$Term_{rel}$+ | $Term_{rel}$~ | { $Term_{scalar}$ -> $Term_{set}$ }

***Definition 2.9** *(AtomicFormula)*

AtomicFormula ::= $Term_{scalar}$ in $Term_{set}$ | $Term_{scalar} = Term_{scalar}$ |
$Term_{set} = Term_{set}$ | $Term_{set}$ <= $Term_{set}$ |
$Term_{rel} = Term_{rel}$ | $Term_{rel}$ <= $Term_{rel}$ |
func $Term_{rel}$ | true | false

***Definition 2.10** *(Wff)*

Wff ::= *AtomicFormula* | not *Wff* | (*Wff* and *Wff*) | (*Wff* or *Wff*)

Definition 2.11 **(Variables of a Term)**

The variables of a term are given by the function, $Var(\tau) : Term \longrightarrow \mathbb{P}\, Variable$, defined as all variables appearing in the term.

Definition 2.12 **(Variables of a Formula)**

The variable of a formula are given by the function, $Var(\phi) : Wff \longrightarrow \mathbb{P}\, Variable$, defined as all variables appearing in the formula.

***Definition 2.13** **(M_{scalar})**

The interpretation of a scalar term τ for an assignment a and a set of values υ is given by the function $M_{scalar}[\tau,\alpha,\upsilon] : Term_{scalar} \times A \times \mathbb{P}\, Value \longrightarrow Value_{scalar} \cup \{$ UNK $\}$, defined as

if τ is v where $v \in Var_{scalar}$
    $\alpha(v)$ if $v \in \mathrm{dom}\ \alpha$
    UNK otherwise

***Definition 2.14** **(M_{set})**

The interpretation of a set term τ for an assignment α and a set of values υ is given by the function $M_{set}[\tau,\alpha,\upsilon] : Term_{set} \times A \times \mathbb{P}\, Value \longrightarrow Value_{set} \cup \{$ UNK $\}$, defined as

if τ is v where $v \in Var_{set}$
    $\alpha(v)$ if $v \in \mathrm{dom}\ \alpha$
    UNK otherwise
if τ is $\{ \tau_1 \}$ where $\tau_1 \in Term_{scalar}$
    $\{ M_{scalar}[\tau_1,\alpha,\upsilon] \}$ if $M_{scalar}[\tau_1,\alpha,\upsilon] \neq$ UNK
    UNK otherwise
if τ is { }
    $\varnothing$
if τ is Un
    $U \cap \upsilon$
if τ is τ_1 U τ_2 where $\tau_1,\tau_2 \in Term_{set}$
    $\{ x \mid x \in M_{set}[\tau_1,\alpha,\upsilon] \vee x \in M_{set}[\tau_2,\alpha,\upsilon] \}$ if $M_{set}[\tau_1,\alpha,\upsilon] \neq$ UNK $\wedge$ $M_{set}[\tau_2,\alpha,\upsilon] \neq$ UNK
    UNK otherwise

if τ is τ_1 & τ_2 where $\tau_1,\tau_2 \in \mathit{Term}_{\mathrm{set}}$

$\{ x \mid x \in M_{\mathrm{set}}[\tau_1,\alpha,\upsilon] \wedge x \in M_{\mathrm{set}}[\tau_2,\alpha,\upsilon] \}$ if $M_{\mathrm{set}}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \wedge M_{\mathrm{set}}[\tau_2,\alpha,\upsilon] \neq \text{UNK}$

UNK otherwise

if τ is $\tau_1 \setminus \tau_2$ where $\tau_1,\tau_2 \in \mathit{Term}_{\mathrm{set}}$

$\{ x \mid x \in M_{\mathrm{set}}[\tau_1,\alpha,\upsilon] \wedge x \notin M_{\mathrm{set}}[\tau_2,\alpha,\upsilon] \}$ if $M_{\mathrm{set}}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \wedge M_{\mathrm{set}}[\tau_2,\alpha,\upsilon] \neq \text{UNK}$

UNK otherwise

if τ is dom τ_1 where $\tau_1 \in \mathit{Term}_{\mathrm{rel}}$

$\{ x \mid \exists y.\ (x,y) \in M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon]\}$ if $M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \neq \text{UNK}$

UNK otherwise

if τ is ran τ_1 where $\tau_1 \in \mathit{Term}_{\mathrm{rel}}$

$\{ y \mid \exists x.\ (x,y) \in M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon]\}$ if $M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \neq \text{UNK}$

UNK otherwise

if τ is $\tau_1(\tau_2)$ where $\tau_1 \in \mathit{Term}_{\mathrm{rel}}$ and $\tau_2 \in \mathit{Term}_{\mathrm{set}}$

$\{ y \mid \exists x.\ x \in M_{\mathrm{set}}[\tau_2,\alpha,\upsilon] \wedge (x,y) \in M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \}$ if $M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \wedge M_{\mathrm{set}}[\tau_2,\alpha,\upsilon] \neq \text{UNK}$

UNK otherwise

***Definition 2.15** **(M_{rel})**

The interpretation of a relational term τ for an assignment α and a set of values υ is given by the function $M_{\mathrm{rel}}[\tau,\alpha,\upsilon] : \mathit{Term}_{\mathrm{rel}} \times A \times \mathbb{P}\mathit{Value} \longrightarrow \mathit{Value}_{\mathrm{rel}} \cup \{ \text{UNK} \}$, defined as

if τ is v where $v \in \mathit{Var}_{\mathrm{rel}}$

$\alpha(v)$ if $v \in \operatorname{dom} \alpha$

UNK otherwise

if τ is $\tau_1 \cup \tau_2$ where $\tau_1,\tau_2 \in \mathit{Term}_{\mathrm{rel}}$

$\{ (x,y) \mid (x,y) \in M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \vee (x,y) \in M_{\mathrm{rel}}[\tau_2,\alpha,\upsilon] \}$ if $M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \wedge M_{\mathrm{rel}}[\tau_2,\alpha,\upsilon] \neq \text{UNK}$

UNK otherwise

if τ is Id

$\{ (x,x) \mid x \in U \}$

if τ is τ_1 & τ_2 where $\tau_1,\tau_2 \in \mathit{Term}_{\mathrm{rel}}$

$\{ (x,y) \mid (x,y) \in M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \wedge (x,y) \in M_{\mathrm{rel}}[\tau_2,\alpha,\upsilon] \}$ if $M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \wedge M_{\mathrm{rel}}[\tau_2,\alpha,\upsilon] \neq \text{UNK}$

UNK otherwise

if τ is $\tau_1 \setminus \tau_2$ where $\tau_1,\tau_2 \in \mathit{Term}_{\mathrm{rel}}$

$\{ (x,y) \mid (x,y) \in M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \wedge (x,y) \notin M_{\mathrm{rel}}[\tau_2,\alpha,\upsilon] \}$ if $M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \wedge M_{\mathrm{rel}}[\tau_2,\alpha,\upsilon] \neq \text{UNK}$

UNK otherwise

if τ is $\tau_1 ; \tau_2$ where $\tau_1,\tau_2 \in \mathit{Term}_{\mathrm{rel}}$

$\{ (x,y) \mid \exists z.\ (x,z) \in M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \wedge (z,y) \in M_{\mathrm{rel}}[\tau_2,\alpha,\upsilon] \}$ if $M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \wedge M_{\mathrm{rel}}[\tau_2,\alpha,\upsilon] \neq \text{UNK}$

UNK otherwise

if τ is τ_1+ where $\tau_1 \in \mathit{Term}_{\mathrm{rel}}$

$\{ (x,y) \mid \exists z_1,z_2,\ldots,z_k.\ (x,z_1) \in M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \wedge (z_1,z_2) \in M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \wedge \ldots \wedge (z_k,y) \in M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \}$ if $M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \neq \text{UNK}$

UNK otherwise

if τ is τ_1~ where $\tau_1 \in \mathit{Term}_{rel}$

$$\begin{array}{ll} \{ (x,y) \mid (y,x) \in M_{rel}[\tau_1,\alpha,\upsilon] \} & \text{if } M_{rel}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \\ \text{UNK} & \text{otherwise} \end{array}$$

if τ is τ_1 <: τ_2 where $\tau_1 \in \mathit{Term}_{set}$ and $\tau_2 \in \mathit{Term}_{rel}$

$$\begin{array}{ll} \{ (x,y) \mid x \in M_{set}[\tau_1,\alpha,\upsilon] \wedge & \text{if } M_{set}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \wedge \\ \qquad (x,y) \in M_{rel}[\tau_2,\alpha,\upsilon] \} & \qquad M_{rel}[\tau_2,\alpha,\upsilon] \neq \text{UNK} \\ \text{UNK} & \text{otherwise} \end{array}$$

if τ is { τ_1 -> τ_2 } where $\tau_1 \in \mathit{Term}_{scalar}$ and $\tau_2 \in \mathit{Term}_{set}$

$$\begin{array}{ll} \{ (x,y) \mid x = M_{scalar}[\tau_1,\alpha,\upsilon] \wedge & \text{if } M_{scalar}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \wedge \\ \qquad y \in M_{set}[\tau_2,\alpha,\upsilon] \} & \qquad M_{set}[\tau_2,\alpha,\upsilon] \neq \text{UNK} \\ \text{UNK} & \text{otherwise} \end{array}$$

*Definition 2.16 (**M_{term}**)

The interpretation of a term τ for an assignment α and a set of values υ is given by the function $M_{term}[\tau,\alpha,\upsilon] : \mathit{Term} \times A \times \mathbf{P}\,\mathit{Value} \longrightarrow \mathit{Value} \cup \{ \text{UNK} \}$, defined by the union of the three specific interpretation functions:

$M_{term} = M_{scalar} \cup M_{set} \cup M_{rel}$

*Definition 2.17 (**M**)

The interpretation of a formula ϕ for an assignment α is given by the function $M[\phi,\alpha,\upsilon] : \mathit{Wff} \times A \times \mathbf{P}\,\mathit{Value} \longrightarrow \{ \text{TRUE, FALSE, UNK} \}$, defined as

if ϕ is $\tau_1 = \tau_2$ where $\tau_1,\tau_2 \in \mathit{Term}_{scalar}$

$$\begin{array}{ll} \text{TRUE} & \text{if } M_{scalar}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \wedge\ M_{scalar}[\tau_2,\alpha,\upsilon] \neq \text{UNK} \wedge \\ & \qquad M_{scalar}[\tau_1,\alpha,\upsilon] = M_{scalar}[\tau_2,\alpha,\upsilon]) \\ \text{FALSE} & \text{if } M_{scalar}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \wedge\ M_{scalar}[\tau_2,\alpha,\upsilon] \neq \text{UNK} \wedge \\ & \qquad M_{scalar}[\tau_1,\alpha,\upsilon] \neq M_{scalar}[\tau_2,\alpha,\upsilon] \\ \text{UNK} & \text{otherwise} \end{array}$$

if ϕ is τ_1 in τ_2 where $\tau_1 \in \mathit{Term}_{scalar}$ and $\tau_2 \in \mathit{Term}_{set}$

$$\begin{array}{ll} \text{TRUE} & \text{if } M_{scalar}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \wedge\ M_{set}[\tau_2,\alpha,\upsilon] \neq \text{UNK} \wedge \\ & \qquad M_{scalar}[\tau_1,a,\upsilon] \in M_{set}[\tau_2,\alpha,\upsilon] \\ \text{FALSE} & \text{if } M_{scalar}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \wedge\ M_{set}[\tau_2,\alpha,\upsilon] \neq \text{UNK} \wedge \\ & \qquad M_{scalar}[\tau_1,\alpha,\upsilon] \notin M_{set}[\tau_2,\alpha,\upsilon] \\ \text{UNK} & \text{otherwise} \end{array}$$

if ϕ is $\tau_1 = \tau_2$ where $\tau_1,\tau_2 \in \mathit{Term}_{set}$

$$\begin{array}{ll} \text{TRUE} & \text{if } M_{set}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \wedge\ M_{set}[\tau_2,\alpha,\upsilon] \neq \text{UNK} \wedge \\ & \qquad M_{set}[\tau_1,\alpha,\upsilon] = M_{set}[\tau_2,\alpha,\upsilon] \\ \text{FALSE} & \text{if } M_{set}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \wedge\ M_{set}[\tau_2,\alpha,\upsilon] \neq \text{UNK} \wedge \\ & \qquad M_{set}[\tau_1,\alpha,\upsilon] \neq M_{set}[\tau_2,\alpha,\upsilon] \\ \text{UNK} & \text{otherwise} \end{array}$$

if ϕ is τ_1 <= τ_2 where $\tau_1,\tau_2 \in \mathit{Term}_{set}$

$$\begin{array}{ll} \text{TRUE} & \text{if } M_{set}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \wedge\ M_{set}[\tau_2,\alpha,\upsilon] \neq \text{UNK} \wedge \\ & \qquad M_{set}[\tau_1,\alpha,\upsilon] \subseteq M_{set}[\tau_2,\alpha,\upsilon] \\ \text{FALSE} & \text{if } M_{set}[\tau_1,\alpha,\upsilon] \neq \text{UNK} \wedge\ M_{set}[\tau_2,\alpha,\upsilon] \neq \text{UNK} \wedge \\ & \qquad \exists x \in M_{set}[\tau_1,\alpha,\upsilon].\ x \notin M_{set}[\tau_2,\alpha,\upsilon] \\ \text{UNK} & \text{otherwise} \end{array}$$

if ϕ is $\tau_1 = \tau_2$ where $\tau_1,\tau_2 \in \mathit{Term}_{\mathrm{rel}}$

TRUE if $M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \neq \mathrm{UNK} \wedge M_{\mathrm{rel}}[\tau_2,\alpha,\upsilon] \neq \mathrm{UNK} \wedge$
$M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] = M_{\mathrm{rel}}[\tau_2,\alpha,\upsilon]$

FALSE if $M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \neq \mathrm{UNK} \wedge M_{\mathrm{rel}}[\tau_2,\alpha,\upsilon] \neq \mathrm{UNK} \wedge$
$M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \neq M_{\mathrm{rel}}[\tau_2,\alpha,\upsilon]$

UNK otherwise

if ϕ is $\tau_1 <= \tau_2$ where $\tau_1,\tau_2 \in \mathit{Term}_{\mathrm{rel}}$

TRUE if $M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \neq \mathrm{UNK} \wedge M_{\mathrm{rel}}[\tau_2,\alpha,\upsilon] \neq \mathrm{UNK} \wedge$
$M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \subseteq M_{\mathrm{rel}}[\tau_2,\alpha,\upsilon]$

FALSE if $M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \neq \mathrm{UNK} \wedge M_{\mathrm{rel}}[\tau_2,\alpha,\upsilon] \neq \mathrm{UNK} \wedge$
$\exists(x,y) \in M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon].\ (x,y) \notin M_{\mathrm{rel}}[\tau_2,\alpha,\upsilon]$

UNK otherwise

if ϕ is func τ_1 where $\tau_1 \in \mathit{Term}_{\mathrm{rel}}$

TRUE if $M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \neq \mathrm{UNK} \wedge$
$\forall x,y,z.((x,y) \in M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \wedge (x,z) \in M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon]) \Rightarrow y = z$

FALSE if $M_{\mathrm{rel}}[\tau_1,a,\upsilon] \neq \mathrm{UNK} \wedge$
$\exists x,y,z.(x,y) \in M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \wedge (x,z) \in M_{\mathrm{rel}}[\tau_1,\alpha,\upsilon] \wedge y \neq z$

UNK otherwise

if ϕ is true

TRUE

if ϕ is false

FALSE

if ϕ is (ϕ_1 and ϕ_2) where $\phi_1,\phi_2 \in \mathit{Wff}$

TRUE if $M[\phi_1,\alpha,\upsilon] = \mathrm{TRUE} \wedge M[\phi_2,\alpha,\upsilon] = \mathrm{TRUE}$

FALSE if $M[\phi_1,\alpha,\upsilon] = \mathrm{FALSE} \vee M[\phi_2,\alpha,\upsilon] = \mathrm{FALSE}$

UNK otherwise

if ϕ is (ϕ_1 or ϕ_2) where $\phi_1,\phi_2 \in \mathit{Wff}$

TRUE if $M[\phi_1,\alpha,\upsilon] = \mathrm{TRUE} \vee M[\phi_2,\alpha,\upsilon] = \mathrm{TRUE}$

FALSE if $M[\phi_1,\alpha,\upsilon] = \mathrm{FALSE} \wedge M[\phi_2,\alpha,\upsilon] = \mathrm{FALSE}$

UNK otherwise

if ϕ is not ϕ_1 where $\phi_1 \in \mathit{Wff}$

TRUE if $M[\phi_1,\alpha,\upsilon] = \mathrm{FALSE}$

FALSE if $M[\phi_1,\alpha,\upsilon] = \mathrm{TRUE}$

UNK otherwise

Definition 2.18 **(Logical Entailment)**

$\phi \models \phi'$ iff $\forall\upsilon \subseteq \mathit{Value}.\forall a \in A.\ M[\phi,a,\upsilon] = \mathrm{TRUE} \Rightarrow M[\phi',a,\upsilon] = \mathrm{TRUE}$

Definition 2.19 **(*Ord*)**

The variables are ordered according to *Ord*, a one-to-one function mapping the variables to the first N natural numbers.

Definition 2.20 **(Var_i)**

$\mathit{Var}_i = \{ v \in \mathit{Variable} \mid 1 \leqslant \mathit{Ord}(v) \leqslant i \}$

Definition 2.21 **(A_i)**

$A_i = \{ \alpha \in A \mid \mathrm{dom}\ \alpha = \mathit{Var}_i \}$

Definition 2.22 **(Solution)**

A full assignment $\alpha \in A_N$ is a *solution* for a formula ϕ and a set of values υ iff

$$M[\phi,\alpha,\upsilon] = \text{TRUE}.$$

Definition 2.23 **(Search)**

A function $\omega(\phi,\upsilon) : \mathit{Wff} \times \mathbf{P}\,\mathit{Value} \longrightarrow A_N$ is a *search* iff

$$\forall \alpha \in \omega(\phi,\upsilon).\ \text{ran}\ \alpha \subseteq \upsilon \wedge M[\phi,\alpha,\upsilon] = \text{TRUE}.$$

Definition 2.24 **(Sound Search)**

A search $\omega(\phi,\upsilon)$ is *sound* for a formula ϕ and a set of values υ iff

$$(\exists a \in A_N.\ M[\phi,\alpha,\upsilon] = \text{TRUE} \wedge \text{ran}\ \alpha \subseteq \upsilon) \Rightarrow \omega(\phi,\upsilon) \neq \emptyset.$$

Definition 2.25 **(Duplication)**

An equivalence relation $\approx_d$ is a *duplication* for the formula ϕ and set of values υ iff

$$\forall \alpha,\alpha' \in A_N.\ \text{ran}\ \alpha \subseteq \upsilon \wedge \text{ran}\ \alpha' \subseteq \upsilon \wedge \alpha \approx_d \alpha' \Rightarrow M[\phi,\alpha,\upsilon] = M[\phi,\alpha',\upsilon]$$

Definition 2.26 **(Partial Assignment Duplication)**

An equivalence relation $\approx_{\rho(\phi')}$ is a *partial assignment duplication* of the related formula ϕ' for a formula ϕ and a set of values υ iff

$$\phi \models \phi' \wedge\ \forall a,a' \in A_N.\ (a \approx_{\rho(\phi')} a' \Leftrightarrow (a = a' \vee (M[\phi',a,\upsilon] = \text{FALSE} \wedge M[\phi',a',\upsilon] = \text{FALSE})))$$

Definition 2.27 **(Duplication Composition)**

For any duplications $\approx_a$ and $\approx_b$ for a formula ϕ and a set of values υ, their composition, $\approx_a \circ \approx_b$, is defined as the smallest relation such that

$$\forall \alpha,\alpha' \in A_N.\ (\ \alpha \approx_a \circ \approx_b \alpha' \Leftrightarrow$$
$$(\ \alpha \approx_a \alpha' \vee \alpha \approx_b \alpha' \vee (\ \exists \alpha'' \in A_N.(\ \alpha \approx_a \circ \approx_b \alpha'' \wedge \alpha'' \approx_a \circ \approx_b \alpha'))))$$

Definition 2.28 **(Generator)**

A function $g : A_{i-1} \times \mathbf{P}\,\mathit{Value} \longrightarrow \mathbf{P}A_i$ is a level i *generator* iff

$$\forall \upsilon \subseteq \mathit{Value}. \forall \alpha \in A_{i-1}.\ \forall \alpha' \in g(\alpha,\upsilon).\ \mathit{Var}_{i-1} \lhd \alpha' = \alpha \wedge \alpha'(v_i) \in (\upsilon \cap \mathit{Typing}(\ v_i)\ )$$

where $v_i \in \mathit{Variable}$ and $\mathit{Ord}(v_i) = i$.

Definition 2.29 **(Sound Generator)**

A level i generator g is *sound* for a duplication $\approx_d$ iff

$$\forall \upsilon \subseteq \mathit{Value}. \forall \alpha \in A_N.\ (M[\phi,\alpha,\upsilon] = \text{TRUE} \wedge \text{ran}\ \alpha \subseteq \upsilon) \Rightarrow$$
$$\exists \alpha' \in A_N.\ \mathit{Var}_i \lhd \alpha' \in g(\mathit{Var}_{i-1} \lhd \alpha,\upsilon) \wedge \alpha \approx_d \alpha' \wedge \text{ran}\ \alpha' \subseteq \upsilon.$$

Definition 2.30 **(*gx*)**

The level i exhaustive-enumeration generator $gx : A_{i-1} \times \mathbf{P}\,\mathit{Value} \longrightarrow \mathbf{P}A_i$ is defined as

$$gx(\alpha,\upsilon) = \{\ \alpha\ \cup \{\ v_i \mapsto x\ \}\ \mid x \in\ \ (\upsilon \cap \mathit{Typing}(\ v_i)\ )\ \}$$

where $v_i \in \mathit{Variable}.\ \mathit{Ord}(v_i) = i$.

Definition 2.31 **(Generator Suite)**

A function $\gamma : \mathit{Variable} \longrightarrow (A \longrightarrow \mathbf{P}A)$ is a *generator suite* iff

$\text{dom}\ \gamma = \mathit{Variable} \wedge \forall v \in \mathit{Variable}.\ \mathit{Ord}(v) = i \Rightarrow \gamma(v)$ is a level i generator.

Definition 2.32 **(Composite Generator)**

A *composite generator* $\gamma^* : \mathbf{P}\,\mathit{Value} \longrightarrow \mathbf{P}A_N$ is defined using a generator suite γ as

$$\gamma^*(\upsilon) = G_N(G_{N-1}(G_{N-2}(\ldots G_2(G_1(A_0,\upsilon),\upsilon)\ldots),\upsilon),\upsilon)$$

where $G_i(Q,\upsilon) = \bigcup_{q \in Q} g_i(q,\upsilon)$ and $g_i = \gamma(v_i)$.

Definition 2.33 **(Selective Enumeration)**

Selective enumeration for the generator suite γ is the procedure that computes the function $\omega^*(\phi,\upsilon)$ for a formula ϕ and a set of values υ, where

$$\omega^*(\phi,\upsilon) = \{ \alpha \in A_N \mid \alpha \in \gamma^*(\upsilon) \wedge M[\phi,\alpha,\upsilon] = \text{TRUE} \}.$$

Definition 2.34 **(Reduction)**

The reduction of a duplication $\approx_d$ for a formula ϕ and a set of values υ is

$$R(\approx_d,\phi,\upsilon) = \frac{|\{ \alpha \in A_N \mid \text{ran } \alpha \subseteq \upsilon \}|}{|\approx_d / \upsilon|}$$

where $|\approx_d / \upsilon|$ is the number of equivalence classes in $\approx_d$ including any assignments containing only values from υ.

Definition 2.35 **(Efficiency)**

The efficiency of a generator suite γ for a duplication $\approx_d$, a formula ϕ, and a set of values υ is

$$E(\gamma,\approx_d,\phi,\upsilon) = \frac{|\approx_d / \upsilon|}{|\gamma^*(\upsilon)|}$$

where $|\approx_d / \upsilon|$ is the number of equivalence classes in $\approx_d$ including any assignments containing only values from υ.

Definition 2.36 **(*ext*)**

The function $ext(Q,\approx_d) : \mathbf{P}A_i \times Duplication \to \mathbf{P}A_N$ is defined by the recursive procedure

```
if Q is empty
        $ext(Q,\approx_d) = \varnothing$
else
        for some $\alpha \in Q$
        $ext(Q,\approx_d) = ext(Q \setminus \{\alpha\},\approx_d) \cup$
                $\{ \alpha' : A_N \mid Var_i \lhd \alpha' = \alpha \wedge \forall \alpha'' \in ext(Q \setminus \{\alpha\},\approx_d).\ \neg\ \alpha' \approx_d \alpha'' \}$
```

Definition 2.37 **(Efficiency of a generator)**

The efficiency of a level *i* generator g for a duplication $\approx_d$, a formula ϕ, and a set of values υ is

$$E(g,\approx_d,\phi,\upsilon) = \frac{\sum_{\alpha \in A_{i-1}} | \{ Var_i \lhd a \mid a \in ext(g(\alpha,\upsilon),\approx_d) \} |}{\sum_{\alpha \in A_{i-1}} | g(\alpha,\upsilon) |}$$

Definition 3.1 **(Short Circuiting Generator)**

A level *i* generator $g : A_{i-1} \times \mathbf{P}Value \to \mathbf{P}A_i$ with an underlying level *i* generator g' and a filter formula ϕ' is a level *i* short-circuiting generator for a formula ϕ and a set of values υ iff

$\phi \models \phi' \wedge Var(\phi') \subseteq Var_i \wedge g(\alpha,\upsilon) = \{ a \mid a \in g'(\alpha,\upsilon) \wedge M[\phi',a,\upsilon] = \text{TRUE} \}$.

Definition 3.2 **(Derived-Variable Generator)**

A level *i* generator $g : A_{i-1} \times \mathbf{P}\mathit{Value} \rightarrow \mathbf{P}A_i$ with a filter formula ϕ' is a level *i* derived-variable generator for a formula ϕ and a set of values υ iff

$\phi \vDash \phi' \wedge \mathit{Var}(\phi') \subseteq \mathit{Var}_i \wedge \forall \alpha \in A_{i-1}.$

$\quad\quad (\exists x \in \upsilon \cap \mathit{Typing}(v_i).(M[\phi', \alpha \cup \{ v_i \mapsto x \}, \upsilon] = \mathrm{TRUE} \wedge$

$\quad\quad\quad\quad \forall x' \in \upsilon \cap \mathit{Typing}(v_i) \setminus \{x\}.M[\phi', \alpha \cup \{ v_i \mapsto x' \}, \upsilon] = \mathrm{FALSE}) \wedge$

$\quad\quad\quad g(\alpha, \upsilon) = \{ \alpha \cup \{ v_i \mapsto x \} \}) \vee$

$\quad\quad (\forall x \in \upsilon \cap \mathit{Typing}(v_i).M[\phi', \alpha \cup \{ v_i \mapsto x \}, \upsilon] = \mathrm{FALSE} \wedge$

$\quad\quad\quad g(\alpha, \upsilon) = \varnothing))$

where $v_i \in \mathit{Variable}$ and $\mathit{Ord}(v_i) = i$.

Definition 3.3 **(Bounded-Generation Generator)**

A level *i* generator $g : A_{i-1} \times \mathbf{P}\mathit{Value} \rightarrow \mathbf{P}A_i$ for a formula ϕ and a set of values υ with a filter formula ϕ', reduced set of values υ', a projection function proj, and an underlying generator g' defined as

$\quad \phi' : \mathit{Wff} . \phi \vDash \phi' \wedge \mathit{Var}(\phi') \subseteq \mathit{Var}_i$

$\quad \upsilon' : \mathbf{P}\mathit{Value} . \upsilon' \subseteq \upsilon$

$\quad \mathrm{proj} : (\upsilon' \cap \mathit{Typing}(v_i)) \times A_{i-1} \mathrm{x}\, \mathbf{P}\mathit{Value} \rightarrow (\upsilon \cap \mathit{Typing}(v_i)) .$

$\quad\quad \forall \alpha \in A_{i-1}. \forall x \in \upsilon \cap \mathit{Typing}(v_i).$

$\quad\quad\quad M[\phi', \alpha \cup \{ v_i \mapsto x \}, \upsilon] = \mathrm{TRUE} \Rightarrow \exists x' \in \upsilon'. x = \mathrm{proj}(x', \alpha, \upsilon)$

$\quad g' : A_{i-1} \times \mathbf{P}\mathit{Value} \rightarrow \mathbf{P}A_i$. g' is a level *i* generator

is a level *i* bounded-generation generator iff

$\quad \forall \alpha \in A_{i-1}. g(\alpha, \upsilon) = \{ \mathrm{proj}(a(v_i), \alpha, \upsilon) \mid a \in g'(\alpha, \upsilon') \}.$

where $v_i \in \mathit{Variable}$ and $\mathit{Ord}(v_i) = i$.

Definition 3.4 **(Limited Soundness)**

A level *i* generator g is *limited sound* for a duplication ≈, a formula ϕ, and a set of values υ under a reduced set of values υ' and a projection function $\mathrm{proj} : (\upsilon' \cap \mathit{Typing}(v_i)) \times A_{i-1} \mathrm{x}\, \mathbf{P}\mathit{Value} \rightarrow (\upsilon \cap \mathit{Typing}(v_i))$ iff

$\upsilon' \subseteq \upsilon \wedge \forall \alpha \in A_N. (M[\phi, \alpha, \upsilon] = \mathrm{TRUE} \wedge \mathrm{ran}\ \alpha a \subseteq \upsilon) \Rightarrow$

$\quad\quad \exists \alpha' \in A_N. \alpha \approx \alpha' \wedge \mathrm{ran}\ \alpha' \subseteq \upsilon \wedge$

$\quad\quad\quad \exists \alpha'' \in g(\mathit{Var}_{i-1} \lhd \alpha, \upsilon'). \mathit{Var}_{i-1} \lhd \alpha' = \mathit{Var}_{i-1} \lhd \alpha'' \wedge$

$\quad\quad\quad\quad \alpha'(v_i) = \mathrm{proj}(\alpha''(v_i), \mathit{Var}_{i-1} \lhd \alpha, \upsilon)$

where $v_i \in \mathit{Variable}$ and $\mathit{Ord}(v_i) = i$.

Definition 3.5 **(Disjunctive Partial Assignment Duplication)**

An equivalence relation $\approx_{\vee(\phi')}$ is a *disjunctive partial-assignment duplication* of the filter formula ϕ' for a formula ϕ and a set of values υ iff

$\phi' \vDash \phi \wedge \forall a, a' \in A_N. (a \approx_{\vee(\phi')} a' \Leftrightarrow (a = a' \vee (M[\phi', a, \upsilon] = \mathrm{TRUE} \wedge M[\phi', a', \upsilon] = \mathrm{TRUE}))).$

Definition 3.6 **(Complete Set Of Values)**

A set of values $\upsilon : \mathbf{P}\mathit{Value}$ is a *complete* set of values for a variable v iff

$$((U \cap \upsilon) \cup \mathbf{P}(U \cap \upsilon) \cup \mathbf{P}((U \cap \upsilon) \times (U \cap \upsilon))) \cap \mathit{Typing}(v) \subseteq \upsilon \subseteq.$$
$$((U \cap \upsilon) \cup \mathbf{P}(U \cap \upsilon) \cup \mathbf{P}((U \cap \upsilon) \times (U \cap \upsilon)))$$

Definition 4.1 **(Permutation)**

A *permutation* $\pi : U \rightarrow U$ is a one-to-one, total mapping of atomic elements.

Definition 4.2 **(Permutations of values)**

A permutation π applied to a value v, given by πv, is defined as

$\pi(v)$ where $v \in \mathit{Value}_{\mathrm{scalar}}$

$\pi(v)$

$\pi(v)$ where $v \in \mathit{Value}_{\mathrm{set}}$

$\{ \pi(x) \mid x \in v \}$

$\pi(v)$ where $v \in \mathit{Value}_{\mathrm{rel}}$

$\{ (\pi(x),\pi(y)) \mid (x,y) \in v \}$

Definition 4.3 **(Permutations of assignments)**

For any permutation π and assignment α, the value of $\pi\alpha$ is defined as

$\pi a = \{ v \mapsto \pi\,(\alpha(v)) \mid v \in \mathrm{dom}\ \alpha \}$

Definition 4.4 **(Product of permutations)**

The product of any two permutations π_1 and π_2, given by $\pi_1\pi_2$, is defined as

$\forall x \in \mathit{Value}.\ \pi_1\pi_2(x) = \pi_2(\pi_1(x))$.

Definition 4.5 **(Swap)**

For any two distinct atomic elements e_0 and e_1, the *swap* denoted by $(e_0\ e_1)$ is the permutation that maps e_0 to e_1, e_1 to e_0, and all other elements to themselves.

Definition 4.6 **(Stabilizing)**

A permutation π *stabilizes* a set s iff $s = \pi s$.

Definition 4.7 **(Isomorphism)**

A one-to-one, total mapping $h{:}\,\mathit{Value} \rightarrow \mathit{Value}$ is an *isomorphism* of *Value* onto itself iff the following conditions hold

$\forall x \in \mathit{Value}_{\mathrm{scalar}}.\ \forall s \in \mathit{Value}_{\mathrm{set}}.\ x \in s \Leftrightarrow h(x) \in h(s)$

$\forall r \in \mathit{Value}_{\mathrm{rel}}.\ \forall t \in \mathit{Value}_{\mathrm{set}}.\ \forall s \in \mathit{Value}_{\mathrm{set}}.\ r(t) = s \Leftrightarrow (h(r))(h(t)) = h(s)$.

Definition 4.8 **(Isomorph Duplication)**

The isomorph duplication $\approx_\pi$ is defined as $\forall a,a' \in A_N.\ a \approx_\pi a' \Leftrightarrow \exists\pi.\ \pi a = a'$.

Definition 4.9 **(Automorphism Group)**

The automorphism group of a value, given by $\mathit{Aut}(x) : \mathit{Value} \rightarrow \mathbf{P}\mathit{Permutation}$, is defined as $\forall x \in \mathit{Value}.\ \mathit{Aut}(x) = \{ \pi \mid \pi x = x \}$

Definition 4.10 **(Automorphism Group for Assignments)**

The automorphism group of an assignment, given by $\mathit{Aut}(\alpha) : A \rightarrow \mathbf{P}\mathit{Permutation}$, is defined as $\forall \alpha \in A.\ \mathit{Aut}(\alpha) = \{ \pi \mid \pi\alpha = \alpha \}$

Definition 4.11 **(Isomorph-Eliminating Generator)**

A level *i* generator $g(\alpha_0,\upsilon) : A_{i-1} \times \mathbf{P}\mathit{Value} \rightarrow \mathbf{P}A_i$ is an *isomorph-eliminating generator* iff

all permutations in $\mathit{Aut}(\alpha_0)$ stabilize υ and

$\forall x \in \upsilon \cap \mathit{Typing}(v_i).\ \exists\alpha \in g(\alpha_0,\upsilon).\ \exists\pi \in \mathit{Aut}(\alpha_0).\ \alpha(v_i) = \pi x$.

Definition 4.12 **(Right coset)**

The right coset G'π where G' is a subgroup of group G and π is an element of G is defined as

$G'\pi = \{ \rho\pi \mid \rho \in G')$

Appendix B: Benchmark Specifications

This appendix presents complete descriptions of each specification in the benchmark suite, along with the complete text. The description page summarizes each schema, operation, and claim along with the scope associated with each test run.

alloc - description

The specification entitled alloc has been used extensively throughout this dissertation to provide examples. It is a simplistic description of a memory allocation scheme, such as malloc. I wrote this example. I include alloc in the suite to provide a simple specification that can be understood thoroughly.

Given Types

Addr	Addresses in the memory system
Value	Values stored in the memory system

Schema Heap

Heap defines the basic structure of the memory heap: usage, a partial function indicating the current state of defined memory and inUse, a set of addresses describing the currently allocated memory. Exactly the addresses that are currently allocated are mapped by usage.

Operation Alloc(a : Addr)

The operation Alloc allocates given by its parameter a. The contents of all previously memory is unchanged after the operation and the address a is now recognized as allocated.

Claim uniqueAddrAlloc

The claim uniqueAddrAlloc states that only currently unallocated addresses are ever allocated. This claim is invalid in all scopes tested. This claim is referred to as uniqueAddr in the tables.

Tested with	#Addr = 3	#Value = 3
Tested with	#Addr = 4	#Value = 4
Tested with	#Addr = 5	#Value = 5

alloc - text

```
[Addr, Data]

Heap =
[
 usage : Addr -> Data
 used : set Addr

|
 /* all currently mapped addresses are used */
 used = dom usage
]

Alloc(addr : Addr) =
[
 Heap

|
 /* Allocating a new address does not change the current allocation */
 used <: usage' = usage

 /* But addr is now mapped (to some unknown data element) */
 used' = used U {addr}
 ]

uniqueAddrAlloc::
[
 Heap
 newAddr : Addr

|
 /* A newly allocated address should not have been in use */
 Alloc(newAddr) => newAddr not in used
]
```

coda - description

The coda specification models an early version of the Coda distributed file system [??]. The original goal of this specification was to demonstrate that the implementation choices made for Coda met the expectations of the abstract model of the distributed file system. This specification is a fragment of one developed by Josh Raiff while working on the project. The complete specification, consisting of over one thousand lines of NP, is the largest specification analyzed by Nitpick or Ladybug. This smaller excerpt includes the second largest claims (by number of variables and number of atomic formulae) checked in the suite.

Given Types

VOL	Volumes in the volume package (concrete only).
INODE	Standard Inodes for file system.
VNODE	Coda Vnodes, representatives for volumes.
NAME	The name of the entry. dot and dotdot are reserved as special names.
ENTRY	Directory entries, mapped by inodes, references a name and a vnode.

Schema VOL_C

This schema, giving the concrete model of the volume management, models the variables implemented in Coda.

Operation ViceCreate(v:VOL,dvn:VNODE,cvn:VNODE,n:NAME, e:ENTRY)

This operation models the concrete action of creating a new volume v named n with vnode cvn in the parent volume indicated by the vnode dvn.

Operation ViceRenameCommon(old_dir:VNODE,new_dir:VNODE,s_vn:VNODE,t_vn:VNODE, old_name:NAME,new_name:NAME, e:ENTRY)

This operation implements the common portion of the concrete renaming action. In the full Coda specification, this operation was shared between several specific renaming operations. This operation renames an item named old_name having entry e and vnode s_vn in old_dir to new_name having vnode t_vn in new_dir.

Operation RenameSameDir(dir:VNODE, s_vn:VNODE, t_vn:VNODE, old_name:NAME, new_name:NAME, e:ENTRY)

This operation models the concrete action of renaming item old_name to new_name, leaving it in the same directory. Uses ViceRenameCommon to implement the actual renaming.

Schema VOL_A

This schema is the abstract model of Coda volume management, representing an idealized view of the system.

Operation Create(dir:ENTRY,file:ENTRY,vn:VNODE)

This abstract operation creates the item file with vnode vn in directory dir.

Operation Rename(old_dir:ENTRY, new_dir:ENTRY, object:ENTRY)

This operation models the abstract action of renaming (moving) the item object from the directory old_dir to the directory new_dir.

Schema AF

This schema is the abstraction function that maps from the concrete to the abstract model.

Claim RCreate

Does the concrete create operation implement the abstract model? This claim is valid.

Tested with #ENTRY=3 #INODE=3 #VNODE=3 #VOL=3 #NAME=3

Claim RSDRefineRename

Does RenameSameDir implement Rename? This claim, referred to as RSDRefRen, is valid.

Tested with #ENTRY=3 #INODE=3 #VNODE=3 #VOL=3 #NAME=3

coda - text

```
[VOL, VNODE, INODE,  ENTRY]

/* VOL    - Volumes in the volume package. (concrete model only)
 * VNODE - Coda Vnodes, objects in the abtract model.
 * INODE- Inode's.  Note that there are two types of inodes.  Directory inodes
 *            and file inodes.  Directory inodes are stored in the volume package's
 *            RVM space and file inodes are part of the file system on the server.
 *            I model directory inodes and keep track of file inodes. (concrete model only)
 * ENTRY- Directory entries in the concrete model, Directory structure in the the
 *                      abstract model.
 */

NAME == {dot, dotdot, ...}


/**************************************************************************************
 * CONCRETE MODEL
 **************************************************************************************/

VOL_C = [
   /* vnodes correspond to files or directories
       all vnodes are kept in the RVM area.
       this syntax says files and dirs are
       sets of VNODEs, and they are disjoint and
       exclusive, and their values don't change
 */
 const symlinks, files, dirs: kind part VNODE

 /* A directory vnode points to an inode.
  */
 inode: inj VNODE -> INODE

 /* vnodes are partitioned into volumes */
 vol: VNODE -> VOL

 /* not all vnodes are in use */
 alloc_vn: set VNODE

 /* not all volumes are in use */
 alloc_vol: set VOL

 /* Not all inodes are in use */
 alloc_inode: set INODE

 /* Directory entries.  A file vnode that has that more than 1 entry pointing to it
  * has hard links to it. */
 /* dir_ent :  INODE ->  P(NAME x VNODE)*/
 entry :  ENTRY -> INODE
 ent_name: ENTRY -> NAME
 ent_vn:  ENTRY -> VNODE
 alloc_ent : set ENTRY

 /* Define a vnode parent function for conveinence.  This is a derived function. */
 vparent: VNODE -> VNODE
|
 /* any vnode assigned to a volume is in use */
 dom vol = alloc_vn
 ran vol = alloc_vol
 dom inode = alloc_vn & dirs
 ran inode = alloc_inode

 /* Allocated entries must be complete */
 dom entry = alloc_ent
 dom ent_name = alloc_ent

 /* If it doesnt have a vn, then it better be a volume root */
 dom ent_vn <= alloc_ent
 ran entry <= inode.dirs
 ran ent_vn <= alloc_vn

 /* dot and dotdot are always directories */
 ran (dom (ent_name :> {dot, dotdot}) <: ent_vn) <= dirs

 /* No directory has duplicate name, for do I say this? */

 /* define parent function on vnodes, this is derived for convienience.  */
 vparent = ent_vn~ ; (dom (ent_name ;> {dot, dotdot}) <: entry) ; inode~

 /* Vnodes that are related to each other are on the same volume */
 vol~ ; ent_vn~ ; entry ; inode~ ; vol <= Id

 vparent+ & Id = {}
]


ViceCreate(v:VOL; dvn:VNODE; cvn:VNODE; n:NAME; e:ENTRY) = [
 VOL_C
|
 /* The Volume is allocated */

 /* v: alloc_vol */  /* not nec, cos of invariant */

 /* The directory is in the volume */
 vol.dvn = v

 /* Parent is a directory */
 dvn : dirs

 /* The parent is allocated */
 dvn : alloc_vn

 /* Child is a file */
 cvn: files

 /* legal name */
 n !: {dot, dotdot}

 /* The name is not elready in the directory */
 n !: ran (dom (entry :> {inode.dvn}) <: ent_name)

 /* We have a new entry, and vnode */
 e !: alloc_ent
 cvn !: alloc_vn

 /* Create the new entry */

 /* alloc_ent' = alloc_ent U {e} */

 /* why doesn't commenting out this expression generate counters? */
 entry' = entry /* U {e->inode.dvn} */
 ent_name' = ent_name U {e->n}
 ent_vn' = ent_vn U {e->cvn}

 alloc_vn' = alloc_vn U {cvn}
 vol' = vol U {cvn->v}

 alloc_inode' = alloc_inode
 inode' = inode
]
```

```
ViceRenameCommon(old_dir: VNODE; new_dir:VNODE; s_vn:VNODE; t_vn:VNODE;
                              old_name:NAME; new_name:NAME; e:ENTRY) = [
 /* old_dirThe directory old_name is in
  * new_dirWhere to move it to (can = old_dir)
  * s_vn old_name's vnode
  * t_vn  new_name's vnode (if it exists, IE is in alloc_vn)
  * old_nameThe name to be renamed
  * new_nameThe name to rename (or move) to
  * e               old_name's entry.
  */
 VOL_C
|
 /* We are dealing with allocated vnodes */
 {old_dir, new_dir} <= (alloc_vn & dirs)
 s_vn : alloc_vn

 /* No cross-volume renames, also volumes are allocated */
 vol.old_dir = vol.new_dir

 /* Leave . and .. alone */
 old_name !: {dot, dotdot}
 new_name !: {dot, dotdot}

 /* A real directory entry, and the one we want */
 e : alloc_ent
 entry.e = inode.old_dir
 ent_name.e = old_name
 ent_vn.e = s_vn

 /* old_dir is s_vn's parent */
 vparent.s_vn = old_dir

 /* Don't create any loops */
 old_dir != t_vn
 new_dir != t_vn
 s_vn != t_vn
 vol' = vol
 inode' = inode
]

/* RenameSameDir - Rename in the same directory */
RenameSameDir(dir: VNODE; s_vn:VNODE; t_vn:VNODE; old_name:NAME;
                              new_name:NAME; e:ENTRY) = [
 /* dirThe directory old_name is in
  * s_vn old_name's vnode
  * t_vn  new_name's vnode (if it exists, IE is in alloc_vn)
  * old_nameThe name to be renamed
  * new_nameThe name to rename (or move) to
  * e               old_name's entry.
  */
 ViceRenameCommon(dir, dir, s_vn, t_vn, old_name, new_name, e)
|
 /* Target can't exist */
 t_vn !: alloc_vn

 /* New_name is not in the directory */
 new_name !: ran (dom (entry :> {inode.dir}) <: ent_name)

 /* Just change the entry's name, not quite what happens in code */
 ent_name' = ent_name (+) {e->new_name}
 ent_vn' = ent_vn
 entry' = entry
]
```

```
/*****************************************************************************************
 * ABSTRACT MODEL
 *****************************************************************************************/

VOL_A = [
  /* NOTE: The abstract model has no concept of volume.  Is this a problem?  Probably
   *      if I put mountpoints into the concrete model.
   */

  /* There are 3 types of objects */
  const symlinks, files, dirs : kind part VNODE

  /* Each directory node is associated with a vnode */
  obj : ENTRY -> VNODE

  /* Parent is the directory tree */
  parent : ENTRY->ENTRY

  /* Maybe not needed */
  vparent : VNODE -> VNODE

  /* Not all nodes are in use */
  alloc_ent : set ENTRY

  /* Note all objects are in use */
  alloc_vn: set VNODE
|
  /* Contrain obj to be defined over allocated nodes */
  dom obj = alloc_ent
  ran obj = alloc_vn

  /* Contrain parent to be a subset of allocated directory nodes.  Actually all but the
   * root will have a parent.
   */
  dom parent <= alloc_ent
  ran parent <= alloc_ent

  /* Define the parent function on vnodes */
  vparent = (obj~ ; parent ; obj)

  /* You can't be your own parent, there are no loops */
  parent+ & Id = {}
]


Create(dir:ENTRY; file:ENTRY; vn:VNODE) = [
  VOL_A
|
  /* We are creating a new file in an existing directoy */
  dir : alloc_ent
  file !: alloc_ent

  /* Since it's a new file */
  vn !: alloc_vn

  /* Make sure we have the right types */
  obj.dir : dirs
  vn : files

  obj' = obj U {file -> vn}
  parent' = parent U {file -> dir}

]
```

```
/* In the abstract world, rename is just move */

Rename(old_dir:ENTRY; new_dir:ENTRY; object:ENTRY) = [
 VOL_A
|
 /* Type checks */
 {old_dir, new_dir, object} <= alloc_ent
 {obj.old_dir, obj.new_dir} <= dirs

 /* Object is in old_dir */
 parent.object = old_dir

 /* Update the state space */
 obj' = obj
 parent' = parent (+) {object -> new_dir}
]


/**************************************************************************************
 * Define an Abstraction function
 **************************************************************************************/

AF = [
 VOL_C
 VOL_A
|
 parent = entry ; inode~ ; ent_vn~
 obj = ent_vn
]


RCreate(v:VOL; dvn:VNODE; cvn:VNODE; n:NAME; dir:ENTRY; file:ENTRY) ::
[AF | dvn = obj.dir and ViceCreate(v, dvn, cvn, n, file) => Create(dir, file, cvn)]


/* RenameSameDir Refines Rename */
RSDRefinesRename(dir:VNODE; s_vn:VNODE; t_vn:VNODE;
      old_name:NAME; new_name:NAME; obj_e:ENTRY; nd_e:ENTRY; od_e:ENTRY) ::
[AF | RenameSameDir(dir, s_vn, t_vn, old_name, new_name, obj_e) =>
       Rename(od_e, nd_e, obj_e)]
```

digicash - description

This specification, originally developed by Daniel Jackson, tracks the flow of digital cash in one model of electronic commerce. This simple model considers only one bank, one merchant, and one customer. I have included this specification as a real-world based specification that is small enough to scale in scope.

Given Types

COIN	The digital cash that is supplied to the customer.
BCOIN	The mirror of the digital cash that the bank maintains.
SIG	A key that can be used to validate the digital cash.

Schema Bank

The bank defines the validation function that generates keys for digital cash and the records of cash that has been issued or used.

Schema Customer

The customer tracks cash currently held (in cholds) and spent (in spent). The coin that the customer holds maps back to the bank's version.

Schema Merchant

The merchant records (in mholds) what digital cash it has received along with the authorization keys obtained by the bank for the transaction.

Schema OnlyValidUsed

This schema describes the property that only validated cash is accepted.

Schema NoSecondSpending

This schema describes the property that a given coin is not spent.

Operation issue(c:Coin)

Issue a new digital coin c from the bank to the customer.

Operation spend(c:Coin,s:SIG)

The customer spends the coin c at the merchant, validated with s.

Operation deposit(b:BCOIN)

Deposit the coin into the bank.

Claim SpendOnce

Check that a coin, once deposited, cannot be spent again. This claim is invalid for all scopes checked

Tested with #COIN = 3 #BCOIN = 3 #SIG = 3

Tested with #COIN = 4 #BCOIN = 4 #SIG = 4

Tested with #COIN = 5 #BCOIN = 5 #SIG = 5

digicash - text

```
[COIN, BCOIN, SIG]

Bank = [
  const valid :  BCOIN <-> SIG
  used :  BCOIN <-> SIG
  issued : set COIN
]

OnlyValidUsed = [Bank | used <= valid]

Customer = [
  blind : tot inj COIN -> BCOIN
  cholds, spent : set COIN
  ]

Merchant = [
  mholds : BCOIN -> SIG
]

issue (c : COIN) = [
  Bank
  Customer
  const Merchant
|
  cholds' = cholds U {c}
  not c in issued
  issued' = issued U {c}
  used' = used
]

spend  (c : COIN ; s: SIG) = [
  const Bank
  const Customer
  Merchant
|
  c in cholds
  mholds' = mholds U {blind.c -> s}
]

deposit  (b : BCOIN) = [
  Bank
  const Customer
  const Merchant
|
  used' = used U {b -> mholds.b}
  mholds.b in valid.{b} \ used.{b}
  b not in dom used
]

UsedAreSpent = [Bank Customer |
  blind.spent = dom used
]

NoSecondSpending (c: COIN) = [Customer | c not in spent]

SpendOnce(b : BCOIN ) :: [Bank Merchant Customer |
   UsedAreSpent and deposit (b) and OnlyValidUsed => NoSecondSpending (blind~.b)
]
```

faa - description

This specification is a portion of one that was developed by Daniel Jackson to check the proof developed to verify the FAA handoff protocol. He simplified the model to consider only the handoff between two controllers for a single flight. This specification includes one valid claim and one invalid claim[1]. This specification is unique in two ways: every variable is set-valued and no isomorph elimination is possible because the two controllers are distinguished.

Given Types

CON Controller — this model has exactly two controllers, named a and b.

Schema State

This schema defines all the variables used in this specification:

ctr : the set of controllers that are currently in charge of the flight.

primary, backup : the set of controllers that have the primary (backup) responsibility for the flight.

primary_up, backup_up : the set of the controllers that have responsibility for the flight if the primary (backup) controller is responsible.

Schema OneController

The important property that exactly one controller has responsibility for the flight at any time.

Schema OnlyAisController

The property that only a is the controller.

Operation Delta()

Indicates all the possible transitions that are allowed.

Operation Op()

Constrains operations to only allow primary_up and backup_up to shrink (or stay the same).

Operation X1b()

Indicate that a has failed as a backup.

Operation X1a()

Indicate that a has failed as a primary.

Claim X1a_check

Check that a failure in a as the primary guarantees a valid handoff. This claim is valid.

Claim x1b_OK

Check that a failure in a as the backup guarantees a valid handoff. This claim is invalid.

Tested with #CON = 2

[1] Although the safety proof for the handoff protocol was flawed, the protocol itself is safe.

faa - text

```
/*
Simplifications: - only consider a single flight
*/

[CON]
CON == {a,b}

/* ctr is the set of controllers who think they have control
primary is the set of controllers X that have X.p set
backup is the set of controllers X that have X.b set
primary_up (backup_up) contains X if X.p (X.b) is up
*/

/*  the body of this schema contains the
abstraction function and an assumption
that only single failures occur
*/

State = [
 ctr : set CON
 primary, backup : set CON
 primary_up, backup_up: set CON
|
 ctr =  (primary_up & primary) U (backup \ primary_up)
 primary_up U backup_up = CON
]

/* at most one controller thinks he has control */
OneController = [State | one ctr]

Pr = [State | a in (primary_up & backup_up) => {a} & primary ={a} & backup ]

OnlyAisController = [State | ctr \ {a} = {}]


/* specifies the allowable transitions */
Delta () = [State |
 a in ctr => (ctr' = {a} or ctr' = {} or ctr' = {b})
 ctr = {} => (ctr' = {} or ctr' = {b})
 b in ctr => ctr' = {b}
]

Op () = [State | primary_up' <= primary_up and backup_up' <= backup_up]

X1b () = [Op() |
 backup' = backup \ {a}
 primary' = primary
]

X1a () = [Op() |
 backup' = backup
 primary' = primary \ {a}
]

PreX1b = [State | OneController and Pr and OnlyAisContoller and (a in ctr)]

PostX1b = [State | OneController and OnlyAisControllrt and (a in primary)
    (not b in ctr) and (a in backup_up => not a in backup)]

FS_1 = [State |  (a in backup_up => not a in backup)]

X1a_check() :: [State | (X1a() and PostX1b) => FS_1']

X1b_OK () :: [State | X1b () and PreX1b => Delta() and PostX1b']
```

finder - description

This specification provides a simple model of the Macintosh desktop, focusing on the interaction of aliases and the trashcan. It was written by Daniel Jackson and originally appeared in [JD95]. Chapter 3 uses this specification to provide additional examples of partial-assignment reductions.

Given Types

OBJ Any object that can appear on the desktop.

Schema Finder

This schema describes the basic desktop, with trash and (hard) drive being distinguished objects. All objects are partitioned into being files or folders, as denoted by membership in one of those sets. The dir relationship indicates the folder (or directory) that contains an object. Some objects are aliases; an alias links to another object.

Operation Move(x, to : OBJ)

This operation moves any object x into the folder indicated by to. The first constraint prevents a folder from being moved into one of its children. The second constraint describes how the dir relationship is changed; moving an object into an alias has the effect of moving the object into the folder to which the alias links. This operation has executions.

Tested with #OBJ = 3

Tested with #OBJ = 4

Tested with #OBJ = 5

Claim TrashingWorks

This claim states that moving an object x into an object to that is in the trash has the effect of throwing away x. This claim is invalid with four or more objects, demonstrated by a counterexample when to is an alias for a folder that is not in the trash. The Macintosh finder warns a user who attempts this action.

Tested with #OBJ = 3

Tested with #OBJ = 4

Tested with #OBJ = 5

finder - text

```
[OBJ]

Finder = [
  const drive, trash: OBJ
  const files, folders: set OBJ
  dir, links: OBJ -> OBJ
  trashed, aliases: set OBJ
 |
  {drive, trash} <= folders \ dom dir
  ran dir <= folders
  trashed = dir~+.{trash}
  not drive in trashed U {trash}
  aliases <= files
  aliases = dom links
  links+ & Id = {}
  files & folders = {}
  files U folders = OBJ
]

Move (x, to: OBJ) = [
  Finder
|
  x not in dir*.{to}
  dir' = dir (+) {x -> ((links* ;> aliases).to)}
  links' = links
]

TrashingWorks (x, to: OBJ) ::
[Finder |  Move (x, to) and to in trashed U {trash} => x in trashed']
```

hla - description

The High Level Architecture (or HLA) is a protocol developed by the Defense Modeling and Simulation Office (DMSO) of the U.S. Department of Defense to describe an anticpated worldwide simulation system. Numerous people at Carnegie Mellon University, including myself, have been analyzing portions of the specification to improve the likelihood of smooth interactions between components developed by different vendors. Chapter 8 describes the portion of this work that was done with Ladybug. This specification models the ownership of object attributes by components (or federates) in the simulation and how ownership can be transferred between federates. A more complete description of the pieces of this specification can be found in Chapter 8. This example offers the largest search space of any benchmark specification and is indicative of the kinds of workout that I expect Ladybug to recieve in the "real" world.

Given Types

Attr An attribute that is maintained by the simulation. In the HLA, attributes are the desription of actual values, not the placeholders for the actual values.

Class The type of an object in HLA.

Fed A federate, the HLA term for a discrete portion of the simulation.

OAttr An object attribute, the placeholder for values associated with an object in the simulation. The constraints require that #Attr * #Obj = #OAttr.

Obj An object being simulated.

Schema ObjectCollection

The variables that model the basic HLA state.

Schema SoundOwners

A collection of properties that describes a sound ownership state.

Claim AttrDivNotSoundOwns

Check that the attribute divestiture notification operation preserves the sound ownership properties. This claim is valid. Referenced in the tables as AttrDivNot.

Tested with #Attr = 2 #Class = 1 #Fed = 2 #OAttr = 6 #Obj = 3

Claim AttrAcqNotSoundOwns

Check that the attribute acquisition notification operations preserves the sound ownership properties. This claim is invalid. Referenced in the tables as AttrAcqNot.

Tested with #Attr = 2 #Class = 1 #Fed = 2 #OAttr = 6 #Obj = 3

Claim ConditionaCompleteOwners

Check that an execution of an entire protocol does not lose any object ownership. This claim is valid. Referenced in the tables as CompOwners.

Tested with #Attr = 2 #Class = 1 #Fed = 2 #OAttr = 6 #Obj = 3

hla - text

```
/* Define the basic kinds of entities to consider */
[CLASS, ATTR, FED, OATTR, OBJECT]

/* Define the basic universe */
ObjectCollection = [
Objects: set OBJECT
Object_Attrs: set OATTR
ObjectToClass: tot OBJECT -> CLASS
ClassAttrsToClass: tot ATTR -> CLASS
ObjAttrsToClassAttrs: tot OATTR -> ATTR
ObjAttrsToObject: tot OATTR -> OBJECT
 |
  /* Only object attributes about known objects are of interest */
  Object_Attrs = dom (ObjAttrsToObject :> Objects)
 ]

GoodObjColl = [ ObjectCollection |

ObjectToClass;ClassAttrsToClass~ = ObjAttrsToObject~;ObjAttrsToClassAttrs

(ObjAttrsToObject;ObjAttrsToObject~ & ObjAttrsToClassAttrs;ObjAttrsToClassAttrs~) <= Id

/*  First invariant says that each instance has the attributes
specified by its class (or has the right number of attributes

2nd invariant states that the intersection of the
two equivalence relations on AttrTo Object and ObjAttrsTo
ClassAttributes intersect only when the same object attributes
are the subject, i.e., two object attributes can't be of the
same type and belong to the same object instance  */
]

/* Explicitly defined state */
SimState = [
GoodObjColl
Federates: set FED
Publishing: FED <-> ATTR
Owns: FED <-> OATTR
]

/* Implicitly defined state */
OwnershipInternalState = [
WillingToDivest:FED <-> OATTR
WillingToAccept: FED <-> OATTR
TargetOwners : FED <-> OATTR
]

/* Total state to consider */
ExecutionState =
[SimState
OwnershipInternalState]
```

```
RequestAttrOwnDivestiture(fed?:FED, obj?:OBJECT, targets?:set FED,
                                            oattrs?:set OATTR) =
[
 ExecutionState
 const SimState
|
 fed? in Federates
 ObjAttrsToObject.oattrs? = {obj?}
 obj? in Objects
 /* ({fed?} <: Un :> oattrs?) is the same as {fed?} x oattrs? */
 ({fed?} <: Un :> oattrs?) <= Owns

 WillingToDivest' = WillingToDivest U ({fed?} <: Un :> oattrs?)
 WillingToAccept' = WillingToAccept
 TargetOwners' = TargetOwners U (targets? <: Un :> oattrs?)
]
RequestAttrOwnAssumption(fed?:FED, obj?:OBJECT, oattrs?:set OATTR) =
[
 const ExecutionState
|
 fed? in Federates
 ObjAttrsToObject.oattrs? = {obj?}
 obj? in Objects
 ({fed?} <: Un :> oattrs?);ObjAttrsToClassAttrs <= Publishing
 ({fed?} <: Un :> oattrs?) & Owns = {}
]
RequestAttrOwnAcquisition(fed?:FED, obj?:OBJECT, oattrs?:set OATTR) =
[
 ExecutionState
 const SimState
|
 fed? in Federates
 ObjAttrsToObject.oattrs? = {obj?}
 obj? in Objects
 ({fed?} <: Un :> oattrs?);ObjAttrsToClassAttrs <= Publishing
 ({fed?} <: Un :> oattrs?) & Owns = {}

 WillingToDivest' = WillingToDivest
 WillingToAccept' = WillingToAccept U ({fed?} <: Un :> oattrs?)
 TargetOwners' = TargetOwners
]
AttrOwnDivestNotify(fed?:FED, obj?:OBJECT, oattrs?:set OATTR) =
[
 ExecutionState
 const ObjectCollection
|
 fed? in Federates
 ObjAttrsToObject.oattrs? = {obj?}
 obj? in Objects
 ({fed?} <: Un :> oattrs?) <= Owns
 ({fed?} <: Un :> oattrs?) <= WillingToDivest

 Owns' = Owns \ ({fed?} <: Un :> oattrs?)
 Federates' = Federates
 Publishing' = Publishing
 WillingToDivest' = WillingToDivest \ ({fed?} <: Un :> oattrs?)
 WillingToAccept' = WillingToAccept
 TargetOwners' = TargetOwners
]
```

```
AttrOwnAcquisitionNotify(fed?:FED, obj?:OBJECT, oattrs?:set OATTR) =
[
 ExecutionState
 const ObjectCollection
|
 fed? in Federates
 ObjAttrsToObject.oattrs? = {obj?}
 Owns~.oattrs? = {}
 /* Only look for owners amongst the target owners */
 obj? in Objects
 oattrs? <= TargetOwners.{fed?}
 ({fed?} <: Un :> oattrs?) <= WillingToAccept

 Owns' = Owns U ({fed?} <: Un :> oattrs?)
 Federates' = Federates
 Publishing' = Publishing

 WillingToAccept' = WillingToAccept \ ({fed?} <: Un :> oattrs?)
 WillingToDivest' = WillingToDivest
 TargetOwners' = TargetOwners ;> oattrs?
]

/* Force a non-empty state */
NonEmpty = [SimState |
Publishing != {}
Owns != {}
Federates != {}
]

/* Define any properties of the state */
 NoTwoOwners = [ SimState | fun Owns~ ]

/* Check that the non-empty state allows two owners */
NoTwoOwnersForced = [NonEmpty | fun Owns~]

NoBadOwnedAttrs  =  [SimState | ran Owns <= Object_Attrs  ]

NoBadOwners = [SimState | dom Owns <= Federates]

OwnsOnlyIfPublishes = [SimState | Owns;ObjAttrsToClassAttrs <= Publishing ]

SoundOwners = [
NoTwoOwners
NoBadOwnedAttrs
NoBadOwners
OwnsOnlyIfPublishes]

CompleteOwners  = [SimState | ran Owns = Object_Attrs]

/* Now construct the claims to test */

/* Check that each modifying operation maintains sound ownership */

AttrDivNotSoundOwns(fed:FED, obj:OBJECT, oattrs:set OATTR)::
  SoundOwners and AttrOwnDivestNotify(fed,obj,oattrs) => SoundOwners'

AttrAcqNotSoundOwns(fed:FED, obj:OBJECT, oattrs:set OATTR)::
  SoundOwners and AttrOwnAcquisitionNotify(fed,obj,oattrs) => SoundOwners'
```

```
/*********************************************************************************/
/* Check against protocol executions, not just single operations              */
/*********************************************************************************/
/* Check for complete ownership after a simple conditional divestiture */
ConditionalCompleteOwners(fed1:FED, fed2:FED, targets : set FED, obj:OBJECT,
                                            oattrs1:set OATTR, oattrs2:set OATTR)::
[
 ExecutionState
|
 /* require the case we are interested in */
 not fed2 = fed1 and
 fed2 in targets and
 oattrs2 <= oattrs1 and
 SoundOwners and

 CompleteOwners and
 /* the conditional divestiture of oattrs1, actually divesting oattrs2 */
   (RequestAttrOwnDivestiture(fed1,obj,targets,oattrs1);
    RequestAttrOwnAssumption(fed2,obj,oattrs1);
    RequestAttrOwnAcquisition(fed2,obj,oattrs2);
    AttrOwnDivestNotify(fed1,obj,oattrs2);
    AttrOwnAcquisitionNotify(fed2,obj,oattrs2)) =>
 CompleteOwners'
]
```

hla bridge - description

This specification adds one complication to the ownership properties checked in the previous section: bridges that link two complete simulations (called federations). Chapter 8 presents a more complete description of this specification. I wrote this specification to check the effect of varying topologies of bridges and federations on valid maintenance of the ownership properties. It represents another large, real-world example to test. This test also provides the largest scope considered for any given type in any test in the suite (7 for FED).

Given Types

Attr An attribute that is maintained by the simulation. In the HLA, attributes are the desription of actual values, not the placeholders for the actual values.

Class The type of an object in HLA.

Fed A federate, the HLA term for a discrete portion of the simulation.

Federation A federation is a collection of federates that provide a simulation. Bridges can join multiple federations into a single larger simulation.

Bridge A bridge is a special federate that can join two federations.

Map The mapping used by a bridge federate to map objects from one federation to the other.

OAttr An object attribute, the placeholder for values associated with an object in the simulation. The constraints require that #Attr * #Obj = #OAttr.

Obj An object being simulated.

Schema ObjectCollection

This schema extends the ObjectCollection schema from the previous hla specification to define objects to belong in federation.

Schema BridgeState

This schema defines the state necessary to describe the bridges that connect two or more federations.

Claim CheckObjectMapping

Do the constraints placed on bridges require that an object only appears in one in a federation? This claim is invalid. The tables refer to this claim as ObjMapping.

Tested with #FED=4 #FEDERATION=2 #OBJECT=3 #BRIDGE=2 #MAP=2 #ATTR=1 #OATTR=3 #CLASS =1

Claim CheckAcyclicObjMaps

Does requiring acyclic bridges require that an object only appears in one in a federation? This claim is valid. The tables refer to this claim as AcyclicObjMaps.

Tested with #FED=4 #FEDERATION=2 #OBJECT=3 #BRIDGE=2 #MAP=2 #ATTR=1 #OATTR=3 #CLASS =1

Tested with #FED=7 #FEDERATION=3 #OBJECT=4 #BRIDGE=3 #MAP=3 #ATTR=1 #OATTR=4 #CLASS =1

hla bridge - text

```
/* Define the basic kinds of entities to consider */
[CLASS, ATTR, FED, OATTR, OBJECT,BRIDGE,FEDERATION,MAP]

/* Define the basic universe */

ObjectCollection = [
  Objects: set OBJECT
  FederationObjects : OBJECT -> FEDERATION
  Object_Attrs: set OATTR
  ObjectToClass: tot OBJECT -> CLASS
  ClassAttrsToClass: tot ATTR -> CLASS
  ObjAttrsToClassAttrs: tot OATTR -> ATTR
  ObjAttrsToObject: tot OATTR -> OBJECT
|
   /* Only object attributes about known objects are of interest */
   Object_Attrs = dom (ObjAttrsToObject :> Objects)
   Objects = dom FederationObjects
 ]

GoodObjColl = [
  ObjectCollection
|
  ObjectToClass;ClassAttrsToClass~ = ObjAttrsToObject~;ObjAttrsToClassAttrs
 (ObjAttrsToObject;ObjAttrsToObject~ & ObjAttrsToClassAttrs;ObjAttrsToClassAttrs~) <= Id

/*  First invariant says that each instance has the attributes
specified by its class (or has the right number of attributes

2nd invariant states that the intersection of the
two equivalence relations on AttrTo Object and ObjAttrsTo
ClassAttributes intersect only when the same object attributes
are the subject, i.e., two object attributes can't be of the
same type and belong to the same object instance  */
]

/* Explicitly defined state */
SimState = [
  GoodObjColl
  Federates: set FED
  Federations : FED -> FEDERATION
  Publishing: FED <-> ATTR
  Owns: FED <-> OATTR
|
  Federates = dom Federations
]

/* Implicitly defined state */
OwnershipInternalState = [
  WillingToDivest:FED <-> OATTR
  WillingToAccept: FED <-> OATTR
  TargetOwners : FED <-> OATTR
]

/* Total state to consider */

ExecutionState =
[SimState
OwnershipInternalState]
```

```
/* Allow bridges between federations */

BridgeState =
[
 SimState
 Bridges : set BRIDGE
 Maps : set MAP
 SurrogateFor : FED -> BRIDGE
 MapsFromObject : MAP -> OBJECT
 MapsToObject : MAP -> OBJECT
 ObjectMapping : OBJECT <-> OBJECT
 MapsForBridge : MAP -> BRIDGE
|
 ran SurrogateFor = Bridges
 ran MapsForBridge = Bridges
 dom MapsForBridge = Maps
 dom MapsToObject = Maps
 dom MapsFromObject = Maps

 ObjectMapping =
        (MapsFromObject~ ; MapsToObject) U (MapsToObject~ ; MapsFromObject)
 dom ObjectMapping <= Objects

 /* limitation -- allow only binary bridges, meaning each object mapped only once */
 ((MapsToObject U MapsFromObject);((MapsToObject U MapsFromObject)~)) &
                          (MapsForBridge;(MapsForBridge~)) <= Id

 /* A bridge has one surrogate for each federation it participates in */
 SurrogateFor;SurrogateFor~ & Federations;Federations~ <= Id

 /* A bridge only maps objects into/out of a federation in which it has a surrogate */
 MapsForBridge~; (MapsFromObject U MapsToObject);FederationObjects <=
               SurrogateFor~;Federations

 /* Each object mapping must be across different  federations */
 (FederationObjects~;MapsFromObject~;MapsToObject;FederationObjects) & Id = {}
]


/* Properties about bridges */
ObjectMappedOncePerFederation =
[
 BridgeState
|
 (FederationObjects~ ; (ObjectMapping+\Id) ; FederationObjects) & Id = {}
 ]

 NoBridgeCycles =
 [
  BridgeState
 |
  (((SurrogateFor;SurrogateFor~)\Id);((Federations;Federations~)\Id))+ & Id = {}
 ]


/* Check the bridge properties */

CheckObjectMapping:: BridgeState => ObjectMappedOncePerFederation

CheckAcyclicObjMaps:: NoBridgeCycles => ObjectMappedOncePerFederation
```

math - description

These claims, part of a larger collection originally encoded by Daniel Jackson, transcribe some mathematical theorems into NP. An incorrect initial encoding of the shroder equivalence has been retained in the specification to include a claim with counterexamples. This specification extends the suite by incorporating some minimally specified problems. These claims allow little or no partial-assignment reductions, depending almost completely on isomorph elimination to make the search tractable.

Given Types

T the basic element type in all the formulae

Claim connex

Check the behavior of universal relations.

Tested with #T = 3

Tested with #T = 4

Tested with #T = 5

Claim comp

Check the behavior of composition. This claim is valid.

Tested with #T = 3

Tested with #T = 4

Tested with #T = 5

Claim closure

Check the behavior of closure. This claim is valid.

Tested with #T = 3

Tested with #T = 4

Tested with #T = 5

Claim shroder

An erroneous encoding of one of the Shröder equivalences. The corrected equivalence replaces the first equality with a less than or equal to. This claim is invalid.

Tested with #T = 3

Tested with #T = 4

Tested with #T = 5

Claim functions

Check the behavior of functions. This claim is valid.

Tested with #T = 3

Tested with #T = 4

Tested with #T = 5

math - text

```
[T]

/* connex */
connex :: [r : T <-> T | Un <= (r U r~) <=> (Un \ r) <= r~]

/* assoc of composition */
comp :: [p, q, r: T <-> T | p ; (q ; r) = (p ; q) ; r]

/* check a property of closure */
closure :: [ p, q: T <-> T | (p U q)* = (p* ; q)* ; p*]

/* schroder equivalence - incorrect formulation */
schroder :: [p, q, r: T <-> T  | p ; q= r <=> p~ ; (Un \ r) <= (Un \ q)]

/* property of functions */
functions :: [f, g, h: T -> T | f ; g & h = (f & (h ; g~)) ; g]
```

mobile IP - description

This specification, which is described more fully in [JNW99], describes parts of the 1996 version of the mobile IPv6 specification, focusing on checking for acyclicity in the forwarding tables. Yuchung Ng wrote this specification under the guidance of Jeannette Wing. This specification is another "real-world" specification.

Given Types

HOST Hosts are the computing elements that host the mobile computers.

MSG Messages describing updates to the routing table are sent to the hosts.

TS Timestamps allow the messages to be sequenced and expire.

Schema net

This schema describes the basic state of the network.

Operation mh_arrive(h:HOST; m:MSG; t:TS)

This operation models the docking of the mobile computer at host h. The message m is sent to the previous router to notify the change of location and t is the expiration time of m.

Operation update_arrival (m:MSG; keeps: set HOST)

This operation describes the behavior when the message m arrives to update the set of hosts, retaining only keeps.

Schema acyclic_caches

This property describes that no cycles exist in the forwarding caches

Claim loc_update_OK_1 (m:MSG; ks: set HOST)

This claim asserts that the update_arrival operation maintains the acyclic_caches property. This claim is invalid.

Tested with #HOST=3 #MSG=3 #TS=3

mobile IP - text

```
[HOST, MSG, TS]

net = [
  router: HOST
  caching: set HOST
  clock: TS
  caches: HOST -> HOST
  cache_exp_time: HOST -> TS
  updates: set MSG
  from,to,where: MSG -> HOST
  send_time, exp_time: MSG -> TS
  timeorder: tot seq TS
  before: TS -> TS
|
  updates = dom to
  updates = dom from
  updates = dom where
  updates = dom send_time
  updates = dom exp_time
  dom cache_exp_time = dom caches
  exp_time <= send_time; before+
  before = {last timeorder} <; timeorder
  not before = {}
  caches & Id = {}
  (from; from~) & (to; to~) & (send_time; send_time~) <= Id (from~; to) & Id = {}
]

mh_arrive (h:HOST; m:MSG; t:TS) = [net |
  not router = h
  router' = h
  not m in updates
  t in before+.{clock}
  clock' in before+.{clock}
  caching' <= caching
  cache_exp_time' = caching' <: cache_exp_time :> (before+).{clock'}
  caches' = dom(cache_exp_time') <: caches updates' = updates U {m}
  send_time' = send_time U {m -> clock}
  exp_time' = exp_time U {m -> t}
  to' = to U {m -> router}
  from' = from U {m -> h}
  where' = where U {m -> h}
]

update_arrival (m:MSG; keeps: set HOST) = [net |
  clock' in before+.{clock}
  keeps <= caching
  caching' = {to.m} U keeps
  cache_exp_time' =
            caching' <: (cache_exp_time (+) {to.m -> exp_time.m}) :> (before+.{clock'})
  caches' = dom (cache_exp_time') <: (caches (+) {to.m -> where.m}) router' = router
  updates' = updates
  to' = to
  from' = from
  where' = where
  send_time' = send_time
  exp_time' = exp_time
  timeorder = timeorder'
  before = before'
]
```

acyclic_caches = [net | caches+ & Id = {}]

host_move_OK_1 (h:HOST; m:MSG; t:TS) :: [net | acyclic_caches and mh_arrive (h, m, t) => acyclic_caches']

loc_update_OK_1 (m:MSG; ks: set HOST) :: [net | acyclic_caches and update_arrival (m, ks) => acyclic_caches']

phone - description

This specification describes a simple model of a phone system, as presented in Chapter 6, and was originally developed by Daniel Jackson. It offers relatively few opportunities for part-assignment reductions, allowing more detailed consideration of isomorph elimination.

Given Types

Number	A phone number.
Phone	A phone.

Schema Switch

The basic model of the phone system, allowing conference calls.

Operation Join(p : Phone, n : Number)

Add the phone at number n to the call originated by p.

Schema NoTwoCallers

The property that every phone call was originated by exactly one phone.

Schema NoCallersCalled

The property that no phone both originated and answered a phone call.

Claim NoTwoCallersPreserved

Check that Join preserves the NoTwoCallers property. This claim behaves similarly to NoCallersCalledPreserved and is not tested in the benchmark suite.

Claim NoCallersCalledPreserved

Check that Join preserves the NoCallersCalled property. Referenced in the tables as CallersCalledP. This claim is invalid for all scopes checked.

Tested with #Number = 3 #Phone = 3

Tested with #Number = 4 #Phone = 4

Tested with #Number = 5 #Phone = 5

phone - text

```
[Phone, Number ]
Switch =
[
  called : Phone <-> Number
  net : Number <-> Phone
  conns : Phone <-> Phone
|
  func net
  conns = called ; net
]
Join(p : Phone, n : Number ) =
[
   Switch
|
   net' = net
   p in dom called
   not n in ran called
   called' = called U { p -> n }
]
NoTwoCallers =
[
 Switch
|
 fun conns~
]
NoCallersCalled =
[
  Switch
|
  dom conns & ran conns  = {}
]
NoTwoCallersPreserved(p: Phone, n : Number)::
[ |   (Join(p,n) and NoTwoCallers) => NoTwoCallers' ]
NoCallersCalledPreserved(p: Phone, n : Number)::
[ |   (Join(p,n) and NoCallersCalled) => NoCallersCalled' ]
```

styles - description

This description, which appeared originally in [JD96b] models the attribute inheritance in Microsoft Word. The model is simplified to consider a style sheet with a single attribute. This test gives another opportunity to test the ability of Ladybug to scale with scope.

Given Types

Style A style sheet. Style sheets form a hierarchy; each style sheet may inherit attributes from its parent. The style sheet denoted by the variable normal is the top of the hierarchy.

Format An attribute maintained by the style sheet. For example, this element could represent different fonts, styles, or sizes.

Schema StyleSheet

This schema describes the model of a style sheet. The function based is the parent relation on style sheets. The function assoc describes what formatting attribute is associated with each style sheet. The function delta describes how this style sheet varies from its parent.

Operation ChangeParent(s,to: Style)

Change the parent of style sheet s to become to.

Schema XiStyleSheet

The property that the style sheet is unchanged.

Claim FormattingPreserved

Check that changing the parent of a style sheet to a third sheet, then back again leaves the forwarding unchanged. This claim is referred to in the tables as FormattingP and is invalid for all scopes. The counterexample occurs when the child has the same attribute as the third sheet but differs from the parent style sheet. The delta is lost when the child is reparented to the third sheet, with the description now noting that the child and parent agree on the attribute. Reparenting the child back to the original parent maintains this commonality, changing the value of the attribute to that of the parent.

Tested with #Style = 3 #Format = 3

Tested with #Style = 4 #Format = 4

Tested with #Style = 5 #Format = 5

styles - text

```
[Style, Format]


StyleSheet = [
 based: Style -> Style
 const normal: Style
 assoc, delta: Style -> Format
|
 normal not in dom based
 ran based \ {normal} <= dom based
 based+ & Id = {}
{normal} <: assoc = {normal} <: delta
{normal} <; assoc = normal} <; (based ; assoc) (+) delta
]

ChangeParent (s,to: Style) = [
 StyleSheet
|
 s in dom based
 based' = based (+) {s -> to}
 assoc' = assoc
 {s} <; delta' = {s} <; delta
]

XiStyleSheet() = [const StyleSheet]

FormattingPreserved (s, from, to: Style) :: [
 StyleSheet
|
 ({s -> from} <= based and ChangeParent (s, to)  ;  ChangeParent (s, from))
        => XiStyleSheet()
]
```

Appendix C: Consequence Closure Rules

Antecedent	Consequent
S0 = S1	S0 <= S1
R0 = R1	R0 <= R1
S0 < S1	S0 <= S1
R0 < R1	R0 <= R1
{ G0 } <= S1	G0 in S1
not G0 in (S1 U S2)	not G0 in S1
G0 in (S1 & S2)	G0 in S1
not G0 in { G1 }	not G0 = G1
(S1 U S2) <= S0	S1 <= S0
(R1 U R2) <= R0	R1 <= R0
not S0 <= (S1 U S2)	not S0 <= S1
not R0 <= (R1 U R2)	not R0 <= R1
(S1 U S2) = S0	(S0 \ S1) <= S2
(R1 U R2) = R0	(R0 \ R1) <= R2
S0 <= (S1 & S2)	S0 <= S1
R0 <= (R1 & R2)	R0 <= R1
(S0 & S1) = {}	S0 <= (Un \ S1)
(R0 & R1) = {}	S0 <= (Un \ R1)
S0 <= (S1 \ S2)	S0 <= S1
R0 <= (R1 \ R2)	R0 <= R1
S0 <= (S1 \ S2)	S0 <= (Un \ S2)
R0 <= (R1 \ R2)	R0 <= (Un \ R2)
S0 <= (S1 \ S2)	not S0 <= S2

Table C.2: The complete set of single antecedent consequence closure rules used by Ladybug.

Antecedent	Consequent
R0 <= (R1 \ R2)	notR0 <= R2
S0 <= (S1 \ S2)	S2 <= (Un \ S0)
R0 <= (R1 \ R2)	R2 <= (Un \ R0)
R1 <= R0	(dom R1) <= (dom R0)
R1 <= R0	(ran R1) <= (ran R0)
R2 <= (R0 ; R1)	(dom R2) <= (dom R0)
R2 <= (R0 ; R1)	(ran R2) <= (ran R1)
(R0 +) <= R1	R0 <= R1
(R1 $ R2) <= R0	R2 <= R0
R0 <= (S1 <: R2)	R0 <= R2
R0 <= (S1 <; R2)	R0 <= R2
R0 <= (R2 :> S1)	R0 <= R2
R0 <= (R2 ;> S1)	R0 <= R2
R0 <= (S1 <: R2)	(dom R0) <= S1
R0 <= (S1 <; R2)	(dom R0) <= (Un \ S1)
R0 <= (R2 :> S1)	(ran R0) <= S1
R0 <= (R2 ;> S1)	(ran R0) <= (Un \ S1)

Table C.2: The complete set of single antecedent consequence closure rules used by Ladybug.

First Antecedent	Second Antecedent	Consequent
S0 = S1	S1 = S2	S0 = S2
R0 = R1	R1 = R2	R0 = R2
S0 <= S1	S1 <= S2	S0 <= S2
R0 <= R1	R1 <= R2	R0 <= R2
S0 <= S1	S1 <= S0	S0 = S1
R0 <= R1	R1 <= R0	R0 = R1
S0 = (S1 U S2)	S2 <= S1	S0 = S1
R0 = (R1 U R2)	R2 <= R1	R0 = R1
S0 <= (S1 U S2)	S2 <= S1	S0 <= S1
R0 <= (R1 U R2)	R2 <= R1	R0 <= R1
S0 = (S1 & S2)	S1 <= S2	S0 = S1
R0 = (R1 & R2)	R1 <= R2	R0 = R1

Table C.3: The complete set of multiple antecedent consequence closure rules defined by Ladybug. These rules were disabled for the analyses used in this thesis.

Original Formula	Simplified Formula
S0 <= (S1 & S0)	S0 <= S1
R0 <= (R1 & R0)	R0 <= R1
S0 <= S0	true
R0 <= R0	true
S0 = S0	true
R0 = R0	true
G0 = G0	true
{} <= S0	true
{} <= R0	true
S0 <= {}	S0 = {}
R0 <= {}	R0 = {}
Un <= S0	S0 = Un
Un <= R0	R0 = Un
S0 <= Un	true
R0 <= Un	true
S0 <= (S0 U S1)	true
R0 <= (R0 U R1)	true
S0 < S0	false
R0 < R0	false
not S0 = S0	false
not R0 = R0	false
not G0 = G0	false

Table C.4: The complete set of formula simplifying rules used by Ladybug.

Original Term	Simplified Term
S0 \ {}	S0
R0 \ {}	R0
S0 \ Un	{}
R0 \ Un	{}
S0 \ S0	{}
R0 \ R0	{}
S0 U (S1 \ S0)	S0 U S1
R0 U (R1 \ R0)	R0 U R1
S0 U ((S1 \ S0) U S2)	(S0 U (S1 U S2))
R0 U ((R1 \ R0) U R2)	(R0 U (R1 U R2))
S0 & (S1 \ S0)	{}
R0 & (R1 \ R0)	{}
S0 \ (S1 \ S0)	S0
R0 \ (R1 \ R0)	R0
S0 \ (S0 \ S1)	S0 & S1
R0 \ (R0 \ R1)	R0 & R1
S0 U S0	S0
R0 U R0	R0
S0 U (S0 U S1)	S0 U S1
R0 U (R0 U R1)	R0 U R1
S0 & S0	S0
R0 & R0	R0
S0 & (S0 & S1)	S0 & S1
R0 & (R0 & R1)	R0 & R1
Un U S0	Un
Un U R0	Un
Un & S0	S0
Un & R0	R0
{} U S0	S0
{} U R0	R0

Table C.5: The complete set of term simplifying rules used by Ladybug.

Original Term	Simplified Term
{} & S0	{}
{} & R0	{}
dom {}	{}
ran {}	{}
dom Un	Un
ran Un	Un
dom (S0 <: R1)	S0 & (dom R1)
dom (S0 <; R1)	(dom R1) \ S0
ran (R1 :> S0)	S0 & (ran R1)
ran (R1 ;> S0)	(ran R1) \ S0

Table C.5: The complete set of term simplifying rules used by Ladybug.

Appendix D: Full Z Model of Ownership

$[\mathit{CLASS}, \mathit{CLASSATTR}, \mathit{OBJECT}, \mathit{OBJECTATTR}, \mathit{FEDERATE}, \mathit{FEDERATION}, \mathit{BRIDGE}, \mathit{MAP}]$

$\mathit{AttributesToClass} : \mathit{CLASSATTR} \rightarrow \mathit{CLASS}$
$\mathit{privToDeleteObject} : \mathit{CLASS} \rightarrowtail \mathit{CLASSATTR}$

$\mathit{privToDeleteObject}^{\sim} \subseteq \mathit{AttributesToClass}$

ObjectCollection

$\mathit{Objects} : \mathbb{P}\ \mathit{OBJECT}$
$\mathit{FederationObjects} : \mathit{OBJECT} \mathrel{⤔} \mathit{FEDERATION}$
$\mathit{ObjectsToClass} : \mathit{OBJECT} \rightarrow \mathit{CLASS}$
$\mathit{ObjectAttrs} : \mathbb{P}\ \mathit{OBJECTATTR}$
$\mathit{ObjectAttrToObject} : \mathit{OBJECTATTR} \twoheadrightarrow \mathit{OBJECT}$
$\mathit{ObjectAttrToClassAttr} : \mathit{OBJECTATTR} \rightarrow \mathit{CLASSATTR}$

$\mathit{ObjectAttrs} = \mathrm{dom}\ (\mathit{ObjectAttrToObject} \rhd \mathit{Objects})$
$\mathit{Objects} = \mathrm{dom}\ \mathit{FederationObjects}$
$\mathit{ObjectToClass}\ ;\ \mathit{AttributesToClass}^{\sim} = \mathit{ObjectAttrToObject}^{\sim}\ ;\ \mathit{ObjectAttrToClassAttr}$
$(\mathit{ObjectAttrToObject}\ ;\ \mathit{ObjectAttrToObject}^{\sim}) \cap$
$\qquad (\mathit{ObjectAttrToClassAttr}\ ;\ \mathit{ObjectAttrToClassAttr}^{\sim})$
$\qquad \subseteq \mathrm{id}\ \mathit{OBJECTATTR}$

SimulationState

$\mathit{ObjectCollection}$
$\mathit{Federates} : \mathbb{P}\ \mathit{FEDERATE}$
$\mathit{Federations} : \mathit{FEDERATE} \mathrel{⤔} \mathit{FEDERATION}$
$\mathit{Publishing} : \mathit{FEDERATE} \leftrightarrow \mathit{CLASSATTR}$
$\mathit{Owns} : \mathit{FEDERATE} \leftrightarrow \mathit{OBJECTATTR}$

$\mathit{Federates} = \mathrm{dom}\ \mathit{Federations}$

$\mathit{OwnershipInternalState}$

$\mathit{WillingToDivest}: \mathit{FEDERATE} \leftrightarrow \mathit{OBJECTATTR}$

$\mathit{WillingToAccept} : \mathit{FEDERATE} \leftrightarrow \mathit{OBJECTATTR}$

$\mathit{TargetOwners} : \mathit{FEDERATE} \leftrightarrow \mathit{OBJECTATTR}$

$\mathit{BridgeState}$

$\mathit{SimulationState}$

$\mathit{Bridges} : \mathbb{P}\ \mathit{BRIDGE}$

$\mathit{SurrogateFor} : \mathit{FEDERATE} \nrightarrow \mathit{BRIDGE}$

$\mathit{ObjectMapping} : \mathit{OBJECT} \leftrightarrow \mathit{OBJECT}$

$\mathit{MapsFromObject} : \mathit{MAP} \longrightarrow \mathit{OBJECT}$

$\mathit{MapsToObject} : \mathit{MAP} \longrightarrow \mathit{OBJECT}$

$\mathit{MapsForBridge} : \mathit{MAP} \longrightarrow \mathit{BRIDGE}$

$\mathrm{ran}\ \mathit{SurrogateFor} = \mathit{Bridges}$

$\mathit{ObjectMapping} = \mathit{MapsFromObject}\sim;\mathit{MapsToObject} \cup \mathit{MapsToObject}\sim;\mathit{MapsFromObject}$

$\mathit{SurrogateFor};\mathit{SurrogateFor}\sim \cap \mathit{Federations};\mathit{Federations}\sim \subseteq \mathrm{id}\ \mathit{FEDERATE}$

$(\mathit{FederationObjects};\mathit{ObjectMapping};\mathit{FederationObjects}\sim) \cap \mathrm{id}\ \mathit{FEDERATION} = \emptyset$

$\mathit{MapsForBridge}\sim;\ (\mathit{MapsFromObject} \cup \mathit{MapsToObject});\mathit{FederationObjects} \subseteq \mathit{SurrogateFor}\sim;\mathit{Federations}$

$\mathit{ExecutionState}$

$\mathit{SimulationState}$

$\mathit{OwnershipInternalState}$

$\mathit{BridgeState}$

NoTwoOwners

SimulationState

$Owns^{\sim} \in (OBJECTATTR \nrightarrow FEDERATE)$

NoBadOwnedAttrs

SimulationState

$\operatorname{ran} Owns \subseteq ObjectAttrs$

NoBadOwners

SimulationState

$\operatorname{dom} Owns \subseteq Federates$

OwnsOnlyIfPublishes

SimulationState

$(Owns \mathbin{;} ObjectAttrToClassAttr) \subseteq Publishing$

CompleteOwners

SimulationState

$ran\ Owns = ObjectAttrs$

SoundDivestments

ExecutionState

$WillingToDivest \subseteq Owns$

SoundAccepts

ExecutionState

$WillingToAccept \cap Owns = \varnothing$

TargetsUnowned

ExecutionState

$\operatorname{ran} TargetOwners \cap \operatorname{ran} Owns = \varnothing$

TObjectMappedOncePerFederation

BridgeState

$\forall o : OBJECT \bullet FederationObjects \rhd ObjectMapping^{*}(\!|o|\!) \subseteq (FEDERATION \nrightarrow OBJECT)$

$$
\begin{array}{l}
\textit{CreateFedExecution} \\
\Delta \mathit{ExecutionState} \\
\mathit{fedex?} : \mathit{FEDERATION} \\
\hline
\mathit{FederationObjects}' \rhd \{\mathit{fedex?}\} = \varnothing \\
\mathit{FederationObjects}' \mathbin{⩤} \{\mathit{fedex?}\} = \mathit{FederationObjects} \\
\mathit{Federations}' \rhd \{\mathit{fedex?}\} = \varnothing \\
\mathit{Federations}' \mathbin{⩤} \{\mathit{fedex?}\} = \mathit{Federations} \\
\\
\mathit{OwnershipInternalState}' = \mathit{OwnershipInternalState} \\
\mathit{Publishing}' = \mathit{Publishing} \\
\mathit{Owns}' = \mathit{Owns}
\end{array}
$$

$$
\begin{array}{l}
\textit{JoinFedExecution} \\
\Delta \mathit{ExecutionState} \\
\mathit{fedex?} : \mathit{FEDERATION} \\
\mathit{fed?} : \mathit{FEDERATE} \\
\hline
\mathit{fed?} \notin \mathit{Federates} \\
\\
\mathit{Federations}' = \mathit{Federations} \cup \{\mathit{fed?} \mapsto \mathit{fedex?}\} \\
\mathit{ObjectCollection}' = \mathit{ObjectCollection} \\
\mathit{OwnershipInternalState}' = \mathit{OwnershipInternalState} \\
\mathit{Publishing}' = \mathit{Publishing} \\
\mathit{Owns}' = \mathit{Owns}
\end{array}
$$

$$
\begin{array}{l}
\textit{RequestAttrOwnDivestiture} \\
\Delta\, \mathit{ExecutionState} \\
\mathit{fed?} : \mathit{FEDERATE} \\
\mathit{targets?} : \mathbb{P}\ \mathit{FEDERATE} \\
\mathit{obj?} : \mathit{Object} \\
\mathit{cattrs?} : \mathbb{P}\ \mathit{CLASSATTR} \\
\mathit{oattrs} : \mathbb{P}\ \mathit{OBJECTATTR} \\
\hline
\mathit{ObjectAttrToClassAtttr} \llparenthesis \mathit{oattrs} \rrparenthesis = \mathit{cattrs?} \\
\mathit{ObjectAttrToObject} \llparenthesis \mathit{oattrs} \rrparenthesis = \{\mathit{obj?}\} \\
\mathit{fed?} \in \mathit{Federates} \\
\{\mathit{fed?}\} \times \mathit{oattrs} \subseteq \mathit{Owns} \\
\\
\mathit{WillingToDivest}' = \mathit{WillingToDivest} \cup (\ \{\mathit{fed?}\} \times \mathit{oattrs}) \\
\mathit{WillingToAccept}' = \mathit{WillingToAccept} \\
\mathit{TargetOwners}' = \mathit{TargetOwners} \cup (\mathit{targets?} \times \mathit{oattrs}) \\
\\
\mathit{SimulationState}' = \mathit{SimulationState}
\end{array}
$$

RequestAttrOwnAssumption

$\Xi ExecutionState$
$fed? : FEDERATE$
$obj? : Object$
$cattrs? : \mathbb{P}\ CLASSATTR$
$oattrs : \mathbb{P}\ OBJECTATTR$

$ObjectAttrToClassAtttr(oattrs) = cattrs?$
$ObjectAttrToObject(oattrs) = \{obj?\}$
$fed? \in Federates$
$(\{fed?\} \times oattrs) \,;\, ObjectAttrToClassAttr \subseteq Publishing$
$(\{fed?\} \times oattrs) \cap Owns = \varnothing$

RequestAttrOwnAcquisition

$\Delta ExecutionState$
$fed? : FEDERATE$
$obj? : Object$
$cattrs? : \mathbb{P}\ CLASSATTR$
$oattrs : \mathbb{P}\ OBJECTATTR$

$ObjectAttrToClassAtttr(oattrs) = cattrs?$
$ObjectAttrToObject(oattrs) = \{obj?\}$
$fed? \in Federates$
$(\{fed?\} \times oattrs) \,;\, ObjectAttrToClassAttr \subseteq Publishing$
$(\{fed?\} \times oattrs) \cap Owns = \varnothing$

$SimulationState' = SimulationState$

$WillingToAccept' = WillingToAccept \cup (\{fed?\} \times oattrs)$
$WillingToDivest' = WillingToDivest$
$TargetOwners' = TargetOwners$

AttrOwnDivestNotify

$\Delta ExecutionState$
$fed? : FEDERATE$
$obj? : Object$
$cattrs? : \mathbb{P}\ CLASSATTR$
$oattrs : \mathbb{P}\ OBJECTATTR$

$ObjectAttrToClassAtttr(oattrs) = cattrs?$
$ObjectAttrToObject(oattrs) = \{obj?\}$
$fed? \in Federates$
$\{fed?\} \times oattrs \subseteq Owns$

$Owns' = Owns \setminus (\{fed?\} \times oattrs)$
$FederationObjects' = FederationObjects$
$Publishing' = Publishing$
$ObjectAttrs' = ObjectAttrs$
$Federates' = Federates$

$WillingToAccept' = WillingToAccept$
$WillingToDivest' = WillingToDivest \setminus (\{fed?\} \times oattrs)$
$TargetOwners' = TargetOwners$

AttrOwnAcquisitionNotify

$\Delta ExecutionState$
$fed? : FEDERATE$
$obj? : Object$
$cattrs? : \mathbb{P}\ CLASSATTR$
$oattrs : \mathbb{P}\ OBJECTATTR$

$ObjectAttrToClassAtttr \llparenthesis oattrs \rrparenthesis = cattrs?$
$ObjectAttrToObject \llparenthesis oattrs \rrparenthesis = \{obj?\}$
$fed? \in Federates$
$Owns\sim \llparenthesis oattrs \rrparenthesis = \varnothing$
$oattrs \subseteq TargetOwners \llparenthesis \{fed?\} \rrparenthesis$

$Owns' = Owns \cup (\{fed?\} \times oattrs)$
$Objects' = Objects$
$Publishing' = Publishing$
$ObjectAttrs' = ObjectAttrs$
$Federates' = Federates$

$WillingToAccept' = WillingToAccept \setminus (\{fed?\} \times oattrs)$
$WillingToDivest' = WillingToDivest$
$TargetOwners' = TargetOwners \unrhd oattrs$

RequestAttrOwnRelease

$\Xi ExecutionState$
$fed? : FEDERATE$
$obj? : Object$
$cattrs? : \mathbb{P}\ CLASSATTR$
$oattrs : \mathbb{P}\ OBJECTATTR$

$ObjectAttrToClassAtttr \llparenthesis oattrs \rrparenthesis = cattrs?$
$ObjectAttrToObject \llparenthesis oattrs \rrparenthesis = \{obj?\}$
$fed? \in Federates$
$(\{fed?\} \times oattrs) = Owns$

$$
\begin{array}{l}
\mathit{PublishObjectClass} \\
\hline
\Delta \mathit{ExecutionState} \\
\mathit{fed?} : \mathit{FEDERATE} \\
\mathit{class?} : \mathit{CLASS} \\
\mathit{cattrs?} : \mathbb{P}\ \mathit{CLASSATTR} \\
\hline
\mathit{AttributesToClass}\llparenthesis \mathit{cattrs?} \rrparenthesis = \{\mathit{class?}\} \land \mathit{cattrs?} = \varnothing \\
\mathit{fed?} \in \mathit{Federates} \\
\mathit{Publishing}' = \mathit{Publishing} \setminus (\ \{\mathit{fed?}\} \lhd \mathit{Publishing} \rhd \mathit{AttributesToClass}^{\sim}\llparenthesis \{\mathit{class?}\} \rrparenthesis) \\
\qquad\qquad\quad \cup (\{\mathit{fed?}\} \times \mathit{cattrs?}) \\
\mathit{Owns}' = \mathit{Owns} \setminus (\{\mathit{fed?}\} \lhd \mathit{Owns} \rhd (\mathit{ObjectAttrToClassAttr} \mathbin{;} \mathit{AttributesToClass})^{\sim}\llparenthesis \{\mathit{class?}\},) \\
\mathit{ObjectCollection}' = \mathit{ObjectCollection} \\
\mathit{OwnershipInternalState}' = \mathit{OwnershipInternalState} \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\mathit{UnpublishObjectClass} \\
\hline
\Delta \mathit{ExecutionState} \\
\mathit{fed?} : \mathit{FEDERATE} \\
\mathit{class?} : \mathit{CLASS} \\
\hline
\mathit{class?} \in \mathit{AttributesToClass}\llparenthesis \mathit{Publishing}\llparenthesis \{\mathit{fed?}\} \rrparenthesis \ \rrparenthesis \\
\mathit{fed?} \in \mathit{Federates} \\
\mathit{Publishing}' = \mathit{Publishing} \setminus (\ \{\mathit{fed?}\} \lhd \mathit{Publishing} \rhd \mathit{AttributesToClass}^{\sim}\llparenthesis \{\mathit{class?}\} \rrparenthesis) \\
\mathit{Owns}' = \mathit{Owns} \setminus (\{\mathit{fed?}\} \lhd \mathit{Owns} \rhd (\mathit{ObjectAttrToClassAttr} \mathbin{;} \mathit{AttributesToClass})^{\sim}\llparenthesis \{\mathit{class?}\},) \\
\mathit{ObjectCollection}' = \mathit{ObjectCollection} \\
\mathit{OwnershipInternalState}' = \mathit{OwnershipInternalState} \\
\hline
\end{array}
$$

Appendix E: NP Specification of Ownership

```
/* Define the basic kinds of entities to consider */
[CLASS, ATTR, FED, OATTR, OBJECT]
/* Define the basic universe */
ObjectCollection = [
   Objects: set OBJECT
   Object_Attrs: set OATTR
   ObjectToClass: tot OBJECT -> CLASS
   ClassAttrsToClass: tot ATTR -> CLASS
   ObjAttrsToClassAttrs: tot OATTR -> ATTR
   ObjAttrsToObject: tot OATTR -> OBJECT
 |
  /* Only object attributes about known objects are of interest */
  Object_Attrs = dom (ObjAttrsToObject :> Objects)
 ]
GoodObjColl = [
   ObjectCollection
|
   ObjectToClass;ClassAttrsToClass~ = ObjAttrsToObject~;ObjAttrsToClassAttrs
  (ObjAttrsToObject;ObjAttrsToObject~ & ObjAttrsToClassAttrs;ObjAttrsToClassAttrs~) <=
Id
/*  First invariant says that each instance has the attributes
specified by its class (or has the right number of attributes
2nd invariant states that the intersection of the
two equivalence relations on AttrTo Object and ObjAttrsToClassAttributes
intersect only when the same object attributes
are the subject, i.e., two object attributes can't be of the
same type and belong to the same object instance  */
]
/* Explicitly defined state */
SimState = [
   GoodObjColl
   Federates: set FED
   Publishing: FED <-> ATTR
   Owns: FED <-> OATTR
]
/* Implicitly defined state */
OwnershipInternalState = [
   WillingToDivest:FED <-> OATTR
   WillingToAccept: FED <-> OATTR
   TargetOwners : FED <-> OATTR
]
/* Total state to consider */
ExecutionState = [ SimState OwnershipInternalState]
```

```
/* Define any properties of the state */
/* Does more than one federate own any object attribute? */
NoTwoOwners = [ SimState | fun Owns~ ]
/* Force a non-empty state */
NonEmpty =
[
  SimState
|
  Publishing  != {}
  Owns != {}
  Federates != {}
]
/* Check that the non-empty state allows two owners */
NoTwoOwnersForced = [NonEmpty | fun Owns~]
/* Check that federates only own valid object attributes */
NoBadOwnedAttrs  =
[
  SimState
|
  ran Owns <= Object_Attrs
]
/* Check that only valid federates own object attributes */
NoBadOwners = [SimState | dom Owns <= Federates]
/* Check that all federates that own object attributes also publish the corresponding class
attribute */
OwnsOnlyIfPublishes =
[
  SimState

|
  Owns;ObjAttrsToClassAttrs <= Publishing
]
/* Check that all required properties hold */
SoundOwners =
[
  NoTwoOwners
  NoBadOwnedAttrs
  NoBadOwners
  OwnsOnlyIfPublishes
]
/* Is every object attribute owned? (May be violated at times) */
CompleteOwners  = [SimState | ran Owns = Object_Attrs]
/* Is every announced willingness to divest for a currently owner attribute */
SoundDivestments = [ExecutionState | WillingToDivest <= Owns]
/* Is any current desire to acquire already satisfied */
SoundAccepts = [ExecutionState | WillingToAccept & Owns = {} ]
/* Does any potential owner already own the attribute */
TargetsUnowned = [ExecutionState | ran TargetOwners & ran Owns = {} ]
```

```
/* Operations defined on the state */

/* Create an empty federation */
CreateFedExecution() =
[
  ExecutionState
  SimState
|
  Objects' = {}
  Object_Attrs' = {}
  Federates' = {}
  Publishing' = {}
  Owns' = {}
  WillingToAccept' = {}
  WillingToDivest' = {}
  TargetOwners' = {}
]

/* Add a new federate to the federation */
JoinFedExecution(fed?:FED) =
[
  ExecutionState
  const ObjectCollection
  const OwnershipInternalState
|
  fed? not in Federates
  Federates' = (Federates U {fed?})
  Publishing = Publishing'
  Owns' = Owns
]
```

```
/* Describe the relevant services */
/* Request Attribute Ownership Divestiture */
RequestAttrOwnDivestiture(fed?:FED, obj?:OBJECT, targets?:set FED,
                                        oattrs?:set OATTR) =
[
 ExecutionState
 const SimState
|
 fed? in Federates
 ObjAttrsToObject.oattrs? = {obj?}
 obj? in Objects
 /* ({fed?} <: Un :> oattrs?) is the same as {fed?} x oattrs? */
 ({fed?} <: Un :> oattrs?) <= Owns
 WillingToDivest' = WillingToDivest U ({fed?} <: Un :> oattrs?)
 WillingToAccept' = WillingToAccept
 TargetOwners' = TargetOwners U (targets? <: Un :> oattrs?)
]

/* Request Attribute ownership Assumption */
RequestAttrOwnAssumption(fed?:FED, obj?:OBJECT, oattrs?:set OATTR) =
[
 const ExecutionState
|
 fed? in Federates
 ObjAttrsToObject.oattrs? = {obj?}
 obj? in Objects
 ({fed?} <: Un :> oattrs?);ObjAttrsToClassAttrs <= Publishing
 ({fed?} <: Un :> oattrs?) & Owns = {}
]

/* Request Attribute Ownership Acquisition */
RequestAttrOwnAcquisition(fed?:FED, obj?:OBJECT, oattrs?:set OATTR) =
[
 ExecutionState
 const SimState
|
 fed? in Federates
 ObjAttrsToObject.oattrs? = {obj?}
 obj? in Objects
 ({fed?} <: Un :> oattrs?);ObjAttrsToClassAttrs <= Publishing
 ({fed?} <: Un :> oattrs?) & Owns = {}
 WillingToDivest' = WillingToDivest
 WillingToAccept' = WillingToAccept U ({fed?} <: Un :> oattrs?)
 TargetOwners' = TargetOwners
]
```

```
/* Attribue Ownership Divestiture Notification */
AttrOwnDivestNotify(fed?:FED, obj?:OBJECT, oattrs?:set OATTR) =
[
 ExecutionState
 const ObjectCollection
|
 fed? in Federates
 ObjAttrsToObject.oattrs? = {obj?}
 obj? in Objects
 ({fed?} <: Un :> oattrs?) <= Owns
 ({fed?} <: Un :> oattrs?) <= WillingToDivest

 Owns' = Owns \ ({fed?} <: Un :> oattrs?)
 Federates' = Federates
 Publishing' = Publishing

 WillingToDivest' = WillingToDivest \ ({fed?} <: Un :> oattrs?)
 WillingToAccept' = WillingToAccept
 TargetOwners' = TargetOwners
]

/* Attribute Ownership Acquisition Notification */
AttrOwnAcquisitionNotify(fed?:FED, obj?:OBJECT, oattrs?:set OATTR) =
[
 ExecutionState
 const ObjectCollection
|
 fed? in Federates
 ObjAttrsToObject.oattrs? = {obj?}
 Owns~.oattrs? = {}

 /* Only look for owners amongst the target owners */
 obj? in Objects
 oattrs? <= TargetOwners.{fed?}
 ({fed?} <: Un :> oattrs?) <= WillingToAccept

 Owns' = Owns U ({fed?} <: Un :> oattrs?)
 Federates' = Federates
 Publishing' = Publishing

 WillingToAccept' = WillingToAccept \ ({fed?} <: Un :> oattrs?)
 WillingToDivest' = WillingToDivest
 TargetOwners' = TargetOwners ;> oattrs?
]

/* Request Attribute Ownership Release */
RequestAttrOwnRelease(fed?:FED, obj?:OBJECT, oattrs?:set OATTR) =
[
 const ExecutionState
|
 fed? in Federates
 ObjAttrsToObject.oattrs? = {obj?}
 obj? in Objects
 ({fed?} <: Un :> oattrs?) <= Owns
]
```

```
/* Publish Object Class */
PublishObjectClass(fed?:FED, class?:CLASS, cattrs?: set ATTR) =
[
 ExecutionState
 const ObjectCollection
 const OwnershipInternalState
|
 ClassAttrsToClass.cattrs? = {class?} or cattrs? = {}
 fed? in Federates

 Federates' = Federates
 Publishing' = Publishing \ ( {fed?} <: Publishing :> (ClassAttrsToClass~.{class?}))
          U ({fed?} <: Un :> cattrs?)
 Owns' = Owns \ ({fed?} <: Owns :> ((ObjAttrsToClassAttrs;ClassAttrsToClass)~.{class?}))
]


/* Unpublish Object Class */
UnpublishObjectClass(fed?:FED, class?:CLASS) =
[
 ExecutionState
 const ObjectCollection
 const OwnershipInternalState
|
 fed? in Federates
 class? in ClassAttrsToClass.(Publishing.{fed?})

 Federates' = Federates
 Publishing' = Publishing \ ( {fed?} <: Publishing :> (ClassAttrsToClass~.{class?}))
 Owns' = Owns \ ({fed?} <: Owns :> ((ObjAttrsToClassAttrs;ClassAttrsToClass)~.{class?}))
]
```

```
/* Now construct the claims to test */

/* Check that each modifying operation maintains sound ownership */
ReqAttrDivSoundOwns(fed:FED, obj:OBJECT, targets:set FED, oattrs:set OATTR)::
  SoundOwners and RequestAttrOwnDivestiture(fed,obj,targets,oattrs) => SoundOwners'

ReqAttrAcqSoundOwns(fed:FED, obj:OBJECT, oattrs:set OATTR)::
  SoundOwners and RequestAttrOwnAcquisition(fed,obj,oattrs) => SoundOwners'

AttrDivNotSoundOwns(fed:FED, obj:OBJECT, oattrs:set OATTR)::
  SoundOwners and AttrOwnDivestNotify(fed,obj,oattrs) => SoundOwners'

AttrAcqNotSoundOwns(fed:FED, obj:OBJECT, oattrs:set OATTR)::
  SoundOwners and AttrOwnAcquisitionNotify(fed,obj,oattrs) => SoundOwners'

PublishSoundOwns(fed:FED, class:CLASS, cattrs:set ATTR)::
  SoundOwners and PublishObjectClass(fed,class,cattrs) => SoundOwners'

UnpublishSoundOwns(fed:FED, class:CLASS)::
  SoundOwners and UnpublishObjectClass(fed,class) => SoundOwners'

/* Check that willing to divest and accept stays sound */

ReqAttrDivSoundDiv(fed:FED, obj:OBJECT, targets:set FED, oattrs:set OATTR)::
  SoundDivestments and RequestAttrOwnDivestiture(fed,obj,targets,oattrs) =>
SoundDivestments'

AttrDivNotSoundDiv(fed:FED, obj:OBJECT, oattrs:set OATTR)::
  SoundDivestments and AttrOwnDivestNotify(fed,obj,oattrs) => SoundDivestments'

ReqAttrAcqSoundAcc(fed:FED, obj:OBJECT, oattrs:set OATTR)::
  SoundAccepts and RequestAttrOwnAcquisition(fed,obj,oattrs) => SoundAccepts'

AttrAcqNotSoundAcc(fed:FED, obj:OBJECT, oattrs:set OATTR)::
  SoundAccepts and AttrOwnAcquisitionNotify(fed,obj,oattrs) => SoundAccepts'
```

```
/*********************************************************************************/
/* Check against protocol executions, not just single operations                */
/*********************************************************************************/

/* Check for complete ownership after a simple conditional divestiture */
ConditionalCompleteOwners(fed1:FED, fed2:FED, targets : set FED, obj:OBJECT,
                                        oattrs1:set OATTR, oattrs2:set OATTR)::
[
 ExecutionState
|
 /* require the case we are interested in */
 not fed2 = fed1 and
 fed2 in targets and
 oattrs2 <= oattrs1 and
 SoundOwners and

 CompleteOwners and
 /* the conditional divestiture of oattrs1, actually divesting oattrs2 */
   (RequestAttrOwnDivestiture(fed1,obj,targets,oattrs1);
    RequestAttrOwnAssumption(fed2,obj,oattrs1);
    RequestAttrOwnAcquisition(fed2,obj,oattrs2);
    AttrOwnDivestNotify(fed1,obj,oattrs2);
    AttrOwnAcquisitionNotify(fed2,obj,oattrs2)) =>
 CompleteOwners'
]


/* How about an unpublish in the middle of an acquisition */
UnpublishInAcquisition(fed:FED, obj:OBJECT, oattr : OATTR)::
[
 ObjectCollection
 const class : CLASS
|
 class = ClassAttrsToClass.(ObjAttrsToClassAttrs.oattr)
 obj = ObjAttrsToObject.oattr

 SoundOwners and
   RequestAttrOwnAcquisition(fed,obj,{oattr});
   UnpublishObjectClass(fed,class);
   AttrOwnAcquisitionNotify(fed,obj,{oattr})
 => SoundOwners'
]


/* Now check that target owners is maintained correctly when ownership is transferred
unconditionally */
UnconditionalSoundTargets(fed1:FED, obj:OBJECT, targets:set FED, oattrs1:set OATTR,
                                        fed2:FED, oattrs2:set OATTR)::
[
|
 /* require the case we are interested in */
 not fed2 = fed1 and
 oattrs2 <= oattrs1 and
 SoundOwners and

 TargetsUnowned and
   (RequestAttrOwnDivestiture(fed1,obj,targets,oattrs1);
    AttrOwnDivestNotify(fed1,obj,oattrs1);
    AttrOwnAcquisitionNotify(fed2,obj,oattrs2)) =>
 TargetsUnowned'
]
```

```
/* And check for conditionally as well */
ConditionalSoundTargets(fed1:FED, obj:OBJECT, targets:set FED, oattrs1:set OATTR,
                                        fed2:FED, oattrs2:set OATTR)::
[
 ExecutionState
|
 /* require the case we are interested in */
 not fed2 = fed1 and
 oattrs2 <= oattrs1 and
 SoundOwners and
 /* the targetowners still owned should be owned by the originating and be willing to divest */
 TargetsUnowned and
    (RequestAttrOwnDivestiture(fed1,obj,targets,oattrs1);
     AttrOwnDivestNotify(fed1,obj,oattrs2);
     AttrOwnAcquisitionNotify(fed2,obj,oattrs2)) =>
 (ran TargetOwners' & ran Owns' = oattrs1 \ oattrs2 and
  dom (Owns' :> (oattrs1 \ oattrs2)) <= { fed1 } and
  {fed1} <: Un :> (oattrs1 \ oattrs2) <= WillingToDivest')
]
```

Appendix F: NP Specification of HLA Bridges

```
/* Define the basic kinds of entities to consider */
[CLASS, ATTR, FED, OATTR, OBJECT]
/* Define the basic universe */
ObjectCollection = [
   Objects: set OBJECT
   FederationObjects : OBJECT -> FEDERATION
   Object_Attrs: set OATTR
   ObjectToClass: tot OBJECT -> CLASS
   ClassAttrsToClass: tot ATTR -> CLASS
   ObjAttrsToClassAttrs: tot OATTR -> ATTR
   ObjAttrsToObject: tot OATTR -> OBJECT
 |
  /* Only object attributes about known objects are of interest */
  Object_Attrs = dom (ObjAttrsToObject :> Objects)
  /* All current objects must be part of some federation */
  Objects = dom FederationObjects
 ]
GoodObjColl = [
   ObjectCollection
 |
   ObjectToClass;ClassAttrsToClass~ = ObjAttrsToObject~;ObjAttrsToClassAttrs
  (ObjAttrsToObject;ObjAttrsToObject~ & ObjAttrsToClassAttrs;ObjAttrsToClassAttrs~) <=
Id
/*  First invariant says that each instance has the attributes
specified by its class (or has the right number of attributes
2nd invariant states that the intersection of the
two equivalence relations on AttrTo Object and ObjAttrsToClassAttributes
intersect only when the same object attributes
are the subject, i.e., two object attributes can't be of the
same type and belong to the same object instance  */
]
```

```
/* Explicitly defined state */
SimState = [
  GoodObjColl
  Federates: set FED
  Federations : FED -> FEDERATION
  Publishing: FED <-> ATTR
  Owns: FED <-> OATTR
]
/* Implicitly defined state */
OwnershipInternalState = [
  WillingToDivest:FED <-> OATTR
  WillingToAccept: FED <-> OATTR
  TargetOwners : FED <-> OATTR
]
/* Allow bridges between federations */
BridgeState =
[
 SimState
 Bridges : set BRIDGE
 Maps : set MAP
 SurrogateFor : FED -> BRIDGE
 MapsFromObject : MAP -> OBJECT
 MapsToObject : MAP -> OBJECT
 ObjectMapping : OBJECT <-> OBJECT
 MapsForBridge : MAP -> BRIDGE
|
 ran SurrogateFor = Bridges
 ran MapsForBridge = Bridges
 dom MapsForBridge = Maps
 dom MapsToObject = Maps
 dom MapsFromObject = Maps
 ObjectMapping =
       (MapsFromObject~ ; MapsToObject) U (MapsToObject~ ; MapsFromObject)
 dom ObjectMapping <= Objects
 /* limitation -- allow only binary bridges, meaning each object mapped only once */
 ((MapsToObject U MapsFromObject);((MapsToObject U MapsFromObject)~)) &
       (MapsForBridge;(MapsForBridge~)) <= Id
 /* A bridge has one surrogate for each federation it participates in */
 SurrogateFor;SurrogateFor~ & Federations;Federations~ <= Id
 /* A bridge only maps objects into/out of a federation in which it has a surrogate */
 MapsForBridge~; (MapsFromObject U MapsToObject);FederationObjects <=
              SurrogateFor~;Federations
 /* Each object mapping must be across different  federations */
 (FederationObjects~;MapsFromObject~;MapsToObject;FederationObjects) & Id = {}
]
/* Total state to consider */
ExecutionState = [ SimState OwnershipInternalState BridgeState]
```

```
/* Define any properties of the state */
/* Does more than one federate own any object attribute? */
NoTwoOwners = [ SimState | fun Owns~ ]
/* Force a non-empty state */
NonEmpty =
[
  SimState
|
  Publishing  != {}
  Owns != {}
  Federates != {}
]
/* Check that the non-empty state allows two owners */
NoTwoOwnersForced = [NonEmpty | fun Owns~]
/* Check that federates only own valid object attributes */
NoBadOwnedAttrs  =
[
  SimState
|
  ran Owns <= Object_Attrs
]
/* Check that only valid federates own object attributes */
NoBadOwners = [SimState | dom Owns <= Federates]
/* Check that all federates that own object attributes also publish the corresponding class
attribute */
OwnsOnlyIfPublishes =
[
  SimState

|
  Owns;ObjAttrsToClassAttrs <= Publishing
]
/* Check that all required properties hold */
SoundOwners =
[
  NoTwoOwners
  NoBadOwnedAttrs
  NoBadOwners
  OwnsOnlyIfPublishes
]
/* Is every object attribute owned? (May be violated at times) */
CompleteOwners  = [SimState | ran Owns = Object_Attrs]
/* Is every announced willingness to divest for a currently owner attribute */
SoundDivestments = [ExecutionState | WillingToDivest <= Owns]
/* Is any current desire to acquire already satisfied */
SoundAccepts = [ExecutionState | WillingToAccept & Owns = {} ]
/* Does any potential owner already own the attribute */
TargetsUnowned = [ExecutionState | ran TargetOwners & ran Owns = {} ]
```

```
/* Properties about bridges */
ObjectMappedOncePerFederation =
[
 BridgeState
|
 (FederationObjects~ ; (ObjectMapping+\Id) ; FederationObjects) & Id = {}
]
AcyclicObjectMapping =
[
 BridgeState
|
 MapsForBridge~;(MapsFromObject U MapsToObject);ObjectMapping ;
                  (MapsFromObject~ U MapsToObject~);MapsForBridge <= Id
]
NoBridgeCycles =
[
 BridgeState
|
 (((SurrogateFor;SurrogateFor~)\Id);((Federations;Federations~)\Id))+ & Id = {}
]
```

```
/* Operations defined on the state */

/* Create an empty federation */
CreateFedExecution() =
[
 ExecutionState
 SimState
|
 Objects' = {}
 Object_Attrs' = {}
 Federates' = {}
 Publishing' = {}
 Owns' = {}
 WillingToAccept' = {}
 WillingToDivest' = {}
 TargetOwners' = {}
]

/* Add a new federate to the federation */
JoinFedExecution(fed?:FED) =
[
 ExecutionState
 const ObjectCollection
 const OwnershipInternalState
|
 fed? not in Federates
 Federates' = (Federates U {fed?})
 Publishing = Publishing'
 Owns' = Owns
]
```

```
/* Describe the relevant services */
/* Request Attribute Ownership Divestiture */
RequestAttrOwnDivestiture(fed?:FED, obj?:OBJECT, targets?:set FED,
                                        oattrs?:set OATTR) =
[
 ExecutionState
 const SimState
|
 fed? in Federates
 ObjAttrsToObject.oattrs? = {obj?}
 obj? in Objects
 /* ({fed?} <: Un :> oattrs?) is the same as {fed?} x oattrs? */
 ({fed?} <: Un :> oattrs?) <= Owns
 WillingToDivest' = WillingToDivest U ({fed?} <: Un :> oattrs?)
 WillingToAccept' = WillingToAccept
 TargetOwners' = TargetOwners U (targets? <: Un :> oattrs?)
]


/* Request Attribute ownership Assumption */
RequestAttrOwnAssumption(fed?:FED, obj?:OBJECT, oattrs?:set OATTR) =
[
 const ExecutionState
|
 fed? in Federates
 ObjAttrsToObject.oattrs? = {obj?}
 obj? in Objects
 ({fed?} <: Un :> oattrs?);ObjAttrsToClassAttrs <= Publishing
 ({fed?} <: Un :> oattrs?) & Owns = {}
]


/* Request Attribute Ownership Acquisition */
RequestAttrOwnAcquisition(fed?:FED, obj?:OBJECT, oattrs?:set OATTR) =
[
 ExecutionState
 const SimState
|
 fed? in Federates
 ObjAttrsToObject.oattrs? = {obj?}
 obj? in Objects
 ({fed?} <: Un :> oattrs?);ObjAttrsToClassAttrs <= Publishing
 ({fed?} <: Un :> oattrs?) & Owns = {}
 WillingToDivest' = WillingToDivest
 WillingToAccept' = WillingToAccept U ({fed?} <: Un :> oattrs?)
 TargetOwners' = TargetOwners
]
```

```
/* Attribue Ownership Divestiture Notification */
AttrOwnDivestNotify(fed?:FED, obj?:OBJECT, oattrs?:set OATTR) =
[
 ExecutionState
 const ObjectCollection
|
 fed? in Federates
 ObjAttrsToObject.oattrs? = {obj?}
 obj? in Objects
 ({fed?} <: Un :> oattrs?) <= Owns
 ({fed?} <: Un :> oattrs?) <= WillingToDivest

 Owns' = Owns \ ({fed?} <: Un :> oattrs?)
 Federates' = Federates
 Publishing' = Publishing

 WillingToDivest' = WillingToDivest \ ({fed?} <: Un :> oattrs?)
 WillingToAccept' = WillingToAccept
 TargetOwners' = TargetOwners
]

/* Attribute Ownership Acquisition Notification */
AttrOwnAcquisitionNotify(fed?:FED, obj?:OBJECT, oattrs?:set OATTR) =
[
 ExecutionState
 const ObjectCollection
|
 fed? in Federates
 ObjAttrsToObject.oattrs? = {obj?}
 Owns~.oattrs? = {}

 /* Only look for owners amongst the target owners */
 obj? in Objects
 oattrs? <= TargetOwners.{fed?}
 ({fed?} <: Un :> oattrs?) <= WillingToAccept

 Owns' = Owns U ({fed?} <: Un :> oattrs?)
 Federates' = Federates
 Publishing' = Publishing

 WillingToAccept' = WillingToAccept \ ({fed?} <: Un :> oattrs?)
 WillingToDivest' = WillingToDivest
 TargetOwners' = TargetOwners ;> oattrs?
]

/* Request Attribute Ownership Release */
RequestAttrOwnRelease(fed?:FED, obj?:OBJECT, oattrs?:set OATTR) =
[
 const ExecutionState
|
 fed? in Federates
 ObjAttrsToObject.oattrs? = {obj?}
 obj? in Objects
 ({fed?} <: Un :> oattrs?) <= Owns
]
```

```
/* Publish Object Class */
PublishObjectClass(fed?:FED, class?:CLASS, cattrs?: set ATTR) =
[
 ExecutionState
 const ObjectCollection
 const OwnershipInternalState
|
 ClassAttrsToClass.cattrs? = {class?} or cattrs? = {}
 fed? in Federates

 Federates' = Federates
 Publishing' = Publishing \ ( {fed?} <: Publishing :> (ClassAttrsToClass~.{class?}))
           U ({fed?} <: Un :> cattrs?)
 Owns' = Owns \ ({fed?} <: Owns :> ((ObjAttrsToClassAttrs;ClassAttrsToClass)~.{class?}))
]


/* Unpublish Object Class */
UnpublishObjectClass(fed?:FED, class?:CLASS) =
[
 ExecutionState
 const ObjectCollection
 const OwnershipInternalState
|
 fed? in Federates
 class? in ClassAttrsToClass.(Publishing.{fed?})

 Federates' = Federates
 Publishing' = Publishing \ ( {fed?} <: Publishing :> (ClassAttrsToClass~.{class?}))
 Owns' = Owns \ ({fed?} <: Owns :> ((ObjAttrsToClassAttrs;ClassAttrsToClass)~.{class?}))
]
```

```
/* Now construct the claims to test */

/* Check that each modifying operation maintains sound ownership */
ReqAttrDivSoundOwns(fed:FED, obj:OBJECT, targets:set FED, oattrs:set OATTR)::
  SoundOwners and RequestAttrOwnDivestiture(fed,obj,targets,oattrs) => SoundOwners'

ReqAttrAcqSoundOwns(fed:FED, obj:OBJECT, oattrs:set OATTR)::
  SoundOwners and RequestAttrOwnAcquisition(fed,obj,oattrs) => SoundOwners'

AttrDivNotSoundOwns(fed:FED, obj:OBJECT, oattrs:set OATTR)::
  SoundOwners and AttrOwnDivestNotify(fed,obj,oattrs) => SoundOwners'

AttrAcqNotSoundOwns(fed:FED, obj:OBJECT, oattrs:set OATTR)::
  SoundOwners and AttrOwnAcquisitionNotify(fed,obj,oattrs) => SoundOwners'

PublishSoundOwns(fed:FED, class:CLASS, cattrs:set ATTR)::
  SoundOwners and PublishObjectClass(fed,class,cattrs) => SoundOwners'

UnpublishSoundOwns(fed:FED, class:CLASS)::
  SoundOwners and UnpublishObjectClass(fed,class) => SoundOwners'

/* Check that willing to divest and accept stays sound */

ReqAttrDivSoundDiv(fed:FED, obj:OBJECT, targets:set FED, oattrs:set OATTR)::
  SoundDivestments and RequestAttrOwnDivestiture(fed,obj,targets,oattrs) =>
SoundDivestments'

AttrDivNotSoundDiv(fed:FED, obj:OBJECT, oattrs:set OATTR)::
  SoundDivestments and AttrOwnDivestNotify(fed,obj,oattrs) => SoundDivestments'

ReqAttrAcqSoundAcc(fed:FED, obj:OBJECT, oattrs:set OATTR)::
  SoundAccepts and RequestAttrOwnAcquisition(fed,obj,oattrs) => SoundAccepts'

AttrAcqNotSoundAcc(fed:FED, obj:OBJECT, oattrs:set OATTR)::
  SoundAccepts and AttrOwnAcquisitionNotify(fed,obj,oattrs) => SoundAccepts'

/* Check the bridge properties */
CheckObjectMapping:: BridgeState => ObjectMappedOncePerFederation
CheckMappingAcyclic:: BridgeState => AcyclicObjectMapping
CheckAcyclicObjMaps:: NoBridgeCycles => ObjectMappedOncePerFederation
```

```
/*******************************************************************************/
/* Check against protocol executions, not just single operations              */
/*******************************************************************************/

/* Check for complete ownership after a simple conditional divestiture */
ConditionalCompleteOwners(fed1:FED, fed2:FED, targets : set FED, obj:OBJECT,
                                    oattrs1:set OATTR, oattrs2:set OATTR)::
[
 ExecutionState
|
 /* require the case we are interested in */
 not fed2 = fed1 and
 fed2 in targets and
 oattrs2 <= oattrs1 and
 SoundOwners and
 CompleteOwners and
 /* the conditional divestiture of oattrs1, actually divesting oattrs2 */
   (RequestAttrOwnDivestiture(fed1,obj,targets,oattrs1);
    RequestAttrOwnAssumption(fed2,obj,oattrs1);
    RequestAttrOwnAcquisition(fed2,obj,oattrs2);
    AttrOwnDivestNotify(fed1,obj,oattrs2);
    AttrOwnAcquisitionNotify(fed2,obj,oattrs2)) =>
 CompleteOwners'
]

/* How about an unpublish in the middle of an acquisition */
UnpublishInAcquisition(fed:FED, obj:OBJECT, oattr : OATTR)::
[
 ObjectCollection
 const class : CLASS
|
 class = ClassAttrsToClass.(ObjAttrsToClassAttrs.oattr)
 obj = ObjAttrsToObject.oattr
 SoundOwners and
   RequestAttrOwnAcquisition(fed,obj,{oattr});
   UnpublishObjectClass(fed,class);
   AttrOwnAcquisitionNotify(fed,obj,{oattr})
 =>  SoundOwners'
]

/* Now check that target owners is maintained correctly when ownership is transferred
unconditionally */
UnconditionalSoundTargets(fed1:FED, obj:OBJECT, targets:set FED, oattrs1:set OATTR,
                                    fed2:FED, oattrs2:set OATTR)::
[
|
 /* require the case we are interested in */
 not fed2 = fed1 and
 oattrs2 <= oattrs1 and
 SoundOwners and
 TargetsUnowned and
   (RequestAttrOwnDivestiture(fed1,obj,targets,oattrs1);
    AttrOwnDivestNotify(fed1,obj,oattrs1);
    AttrOwnAcquisitionNotify(fed2,obj,oattrs2)) =>
 TargetsUnowned'
]
```

```
/* And check for conditionally as well */
ConditionalSoundTargets(fed1:FED, obj:OBJECT, targets:set FED, oattrs1:set OATTR,
                                                fed2:FED, oattrs2:set OATTR)::
[
 ExecutionState
|
 /* require the case we are interested in */
 not fed2 = fed1 and
 oattrs2 <= oattrs1 and
 SoundOwners and
 /* the targetowners still owned should be owned by the originating and be willing to divest */
 TargetsUnowned and
    (RequestAttrOwnDivestiture(fed1,obj,targets,oattrs1);
     AttrOwnDivestNotify(fed1,obj,oattrs2);
     AttrOwnAcquisitionNotify(fed2,obj,oattrs2)) =>
 (ran TargetOwners' & ran Owns' = oattrs1 \ oattrs2 and
  dom (Owns' :> (oattrs1 \ oattrs2)) <= { fed1 } and
  {fed1} <: Un :> (oattrs1 \ oattrs2) <= WillingToDivest')
]
```

Bibliography

[ABM98] Paul E. Ammann, Paul E. Black, and William J. Majurski, Using Model Checking to Generate Tests from Specifications, *Proceedings of 2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, Brisbane, Australia,December 1998, pp. 46-54.

[Avi96] David Avis. Generating Rooted Triangulations Without Repetitions, *Algorithmica*, Vol. 16, No. 6, December 1996, pp. 618–632.

[BC+92] J. R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{30} States and Beyond. *Information and Computation*, Vol. 98, No. 2, June 1992, pp. 143–170.

[BF97] Avrim L. Blum and Merrick L. Furst. Fast Planning Through Planning Graph Analysis. *Artificial Intelligence*. Vol. 90, No. 1-2, February 1997, pp. 281–300.

[BFP88] Cynthia A. Brown, Larry Finkelstein, and Paul W. Purdom. Backtrack Searching in the Presence of Symmetry in *Sixth International Conference on Algebraic Algorithms and Error Correcting Codes (AAECC)*, Springer Verlag Lecture Notes in Computer Science, Vol. 357, 1988, pp. 99–110.

[BJR99] Grady Booch, Ivan Jacobsen, and James Rumbaugh. *The Unified Modelling Language for Object-Oriented Development. Documentation Set*, version 1.3, Rational Software Corporation, Available at http://www.rational.com/uml/resources/documentation/index.jtmpl>.

[BK79] Lászlo Babai and Ludek Kucera. Canonical Labelling of Graphs in Linear Average Time. *20th Annual Symposium on Foundations of Computer Science*, October 1979, pp. 39–46.

[Bri96] Gunnar Brinkmann. Fast generation of cubic graphs, *Journal of Graph Theory*, Vol. 23, No. 2, October 1996, pp. 139–149.

[Bry92] Randal E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, Vol. 24, No. 3, September 1992, pp. 293–318.

[BW94] Anthony Barret and Daniel S. Weld. Partial-Order Planning: Evaluating Possible Efficiency Gains. *Artificial Intelligence*. Vol. 67, No. 1, May 1994, pp. 71–112.

[CE+96] Edmund M. Clarke, Reinhard Enders, Thomas Filkorn, and Somesh Jha. Exploiting Symmetry in Temporal Model Checking. *Formal Methods in System Design*, Vol. 9, No. 1-2, August 1996, pp. 77–104.

[CG+96] James Crawford, Matthew L. Ginsberg, Eugene Luks, and Amitabha Roy. Symmetry-Breaking Predicates for Search Problems in *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96)*, November 1996, pp. 148–159.

[Che76] Peter P. Chen. The Entity-Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems,* Vol. 1, No. 1, 1976, pp. 9–36.

[CP96] Ching-Tsun Chou, Doron Peled: Formal Verification of a Partial-Order Reduction Technique for Model Checking. *Proceedings Second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96),* Passau, Germany, *Lecture Notes in Computer Science,* Vol. 1055, Springer Verlag, March 1996, pp. 241–257.

[CPS93] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems. *ACM Transactions on Programming Languages and Systems,* Vol. 15, No. 1, January 1993, pp. 36–72.

[CR79] Charles J. Colbourn and Ronald C. Read. Orderly Algorithms For Generating Restricted Classes of Graphs. *Journal of Graph Theory,* Vol. 3, 1979, pp. 187–195.

[CW+96] Edmund M. Clarke, Jeannette M. Wing, et al. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys,* Vol. 28, No. 4, December 1996, pp. 626–643.

[DF98] Rina Dechter and Daniel Frost. *Backtracking Algorithms for Constraint Satisfaction Problems — a Tutorial Survey.* Technical Report, Department of Information and Computer Science, University of California, Irvine, April 1998.

[DGM94] Jeffrey H. Dinitz, David K. Garnick, and Brendan D. McKay. There are 526,915,620 Nonisomorphic One-factorizations of K12, *Journal of Combinatorial Design,* Vol. 2, 1994, pp. 273–285.

[DJ96] Craig A. Damon and Daniel Jackson. Efficient Search as a Means of Executing Specifications. *Proceedings Second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96),* Passau, Germany, *Lecture Notes in Computer Science,* Vol. 1055, Springer Verlag, March 1996, pp. 70–86.

[DJJ96] Craig A. Damon, Daniel Jackson, and Somesh Jha. Checking Relational Specifications with Binary Decision Diagrams. *Proceedings 4th ACM SIGSOFT Conference on Foundations of Software Engineering,* San Francisco, CA, October 1996, pp. 70–81.

[DM+99] Craig A. Damon, Ralph Melton, Robjert J. Allen, Elizabeth Bigelow, James M.Ivers, and David Garlan, *Formalizing a Specification For Analysis: The HLA Ownership Properties.* Technical Report CMU-CS-99-106, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, January 1999.

[DOD97] Defense Modeling and Simulation Office. *High Level Architecture Interface Specification Version 1.2.* August 13, 1997. http://hla.dmso.mil/tech/.

[DP60] Martin Davis and Hilary Putnam. A Computing Procedure For Quantification Theory. *Jorurnal of the Association of Computing Machinery,* Vol. 3, No. 7, May 1960, pp. 201–215.

[ES94] Marcin Engel and Jens Ulrik Skakkeback. *Applying PVS to Z.* Technical Report ID/DTU ME 3/1, ProCos Project, Department of Computer Science, Technical University of Denmark, Lyngby, Denmark, 1994.

[FHL80] Merrick Furst, John Hopcroft, and Eugene Luks. Polynomial-Time Algorithms for Permutation Groups. *21st Annual Symposium on Foundations of Computer Science,* October 1980, pp. 36–41

[FN71] Richard Fikes and Nils J. Nillson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*. Vol. 2, No. 3–4, 1971, pp. 189–208.

[Fox83] Mark S. Fox. *Constraint Directed Search: A Case Study of Job-Shop Scheduling*. Technical Report CMU-CS-83-161. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, December, 1983.

[Fre91] Eugene C. Freuder. Eliminating Interchangeable Values in Constraint Satisfaction Problems. *Proceedings of the 9th National Conference on Artificial Intelligence*, Vol. 1, July 1991, pp. 227–233.

[GLM95] Thomas Grüner, Reinhard Laue, and Markus Meringer. Algorithms for Group Actions Applied to Graph Generation. *Groups and Computation II, DIMACS series in discrete mathematics and theoretical computer science*, Vol. 28, June 1995, pp 113–122.

[Gol92] Leslie Ann Goldberg. Efficient Algorithms for Listing Unlabeled Graphs, *Journal of Algorithms*, Vol. 13, No. 1, March 1992, pp. 128–143.

[HE80] Robert M. Haralick and Gordon Elliot, Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, Vol. 14, No. 3, October 1980, pp. 263–313.

[Hof81] Christoph M. Hoffman. *Group-Theoretic Algorithms and Graph Isomorphism* . Lecture Notes in Computer Science 136. Springer-Verlag, Berlin ; New York, 1982.

[ID96] C. Norris Ip and David L. Dill. Better Verification Through Symmetry. *Formal Methods in System Design*, Vol. 9, No. 1-2, August 1996., pp. 41–76.

[ITU93] ITU-TS. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)*. ITU-TS, Geneva, September 1993.

[Jac98] Daniel Jackson. An Intermediate Design Language and its Analysis. *Proceedings Foundations of Software Engineering*, Orlando, FL, November 1998, pp. 121–130.

[JD95] Daniel Jackson and Craig A. Damon. *Semi-Executable Specifications*. Technical Report CMU-CS-95-216, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, November 1995.

[JD96a] Daniel Jackson and Craig A. Damon. *Nitpick: A Checker for Software Specifications (Reference Manual)*. Technical Report CMU-CS-96-109, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, January 1996.

[JD96b] Daniel Jackson and Craig A. Damon. Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. *IEEE Transactions on Software Engineering*, Vol. 22, No. 7, July 1996, pp. 484–495.

[Jer86] Mark Jerrum. A Compact Representation for Permutation Groups, Journal of Algorithms, Vol. 7, No. 1, March 1986, pp. 60–78.

[Jha96] Somesh Jha. *Symmetry and Induction in Model Checking*. Technical Report CMU-CS-96-202, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, October, 1996.

[JJD96] Daniel Jackson, Somesh Jha, and Craig A. Damon. Faster Checking of Software Specifications by Eliminating Isomorphs, *Proceedings of ACM Symposium on Principles of Programming Languages (POPL '96)*, St. Petersburg Beach, FL, January 1996, pp. 79–90.

[JJD98] Daniel Jackson, Somesh Jha, and Craig A. Damon. Isomorph-free Model Enumeration: A New Method for Checking Relational Specifications. *ACM Transactions on Programming Languages and Systems*, Vol. 20, No. 2, March 1998, pp. 302–343.

[JNW99] Daniel Jackson, Yuchang Ng, and Jeannette M. Wing. A Nitpick Analysis of Mobile IPv6. *Formal Aspects of Computing*, to appear.

[JV00] Daniel Jackson and Mandana Vaziri. Finding Bugs with a Constraint Solver. *Proceedings International Symposium on Software Testing and Analysis* (ISSTA 2000), Portland, OR, August 2000, to appear.

[Kum92] Vipin Kumar. Algorithms for Constraint Satisfaction, A Survey. *AI Magazine,* Vol. 13, No. 1, Spring 1992, pp. 32–44.

[LP94] Shie-Jue Lee and David A. Plaisted. Problem Solving by Searching for Models with a Theorem Prover. *Artificial Intelligence* . Vol. 69, No 1-2, September 1994, pp. 205–233.

[LT89] Clement W. H. Lam and Larry Thiel. Backtrack Search with Isomorph Rejection and Consistency Check. *Journal of Symbolic Computation,* Vol. 7, No. 5,May 1989, pp. 473–485.

[Mac77] Alan K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence,* Vol. 8, No. 1, February 1977, pp. 99–118.

[Mac92] Alan K. Mackworth. The Logic of Constraint Satisfaction. *Artificial Intelligence,* Vol. 58, No. 1-3, December 1992, pp. 3–20.

[McK81] Brendan D. McKay. Practical Graph Isomorphism. *Congressus Numerantium* 21 (1981), pp. 499–517.

[McK94] Brendan D. McKay. *Nauty User's Guide,* version 1.5. Computer Science Department, Australian National University, GPO Box 4, ACT 2601, Australia.

[McK98] Brendan D. McKay. Isomorph-Free Exhaustive Generation. *Journal of Algorithms,* Vol 26, No. 2, February 1998, pp. 306–324.

[Mil79] Gary L. Miller. Graph Isomorphism, General Remarks. *Journal of Computer and System Sciences,* Vol. 18, No. 2, April 1979, pp. 128–142.

[Miy95] Takunari Miyazaki. The Complexity of McKay's Canonical Labeling Algorithm. *Groups and computation II, DIMACS series in discrete mathematics and theoretical computer science,* Vol. 28, June 1995, pp 239–256.

[Pro93] Patrick Prosser. Hybrid Algorithms for Constraint Satisfaction Problems. *Computational Intelligence.* Vol. 9, No. 3, 1993, pp. 268–299.

[Rea81] Ronald C. Read. A Survey of Graph Generation Techniques, *Combinatorial Mathematics VIII, Lecture Notes in Mathematics,* Vol. 884, Springer-Verlag, 1981, pp. 77–89.

[SB+98] David T. Shen, Christina Bouwens, Wesley Braudaway, and Daphne Hurrell. *Bridge Federate System Requirements Based on High Level Architecture Interface Specification Version 1.3 Draft 9.* Document HLASEA-A001, Science Applications International Corporation, July 2, 1998.

[SF95] Normal M. Sadeh and Mark S. Fox. *Variable and Value Ordering Heuristics for the Job Shop Scheduling Constraint Satisfaction Problem..* Technical Report CMU-RI-TR-95-39, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, 1995.

[Sla94] John K. Slaney. Finder: Finite Domain Enumerator, System Description. *Proceedings of 12th International Conference on Automated Deduction,* Nancy, France, Alan Bundy (ed.), *Lecture Notes in Artificial Intelligence,* Vol. 814, Springer Verlag, Berlin, June 1994, pp. 798–801.

[Sla00] John Slaney. *Finder Finite Domain Enumerator; Notes and Guide.* Version 3.0, Centre for

Information Science Research, Australian National University, January 2000. Available at http://arp.anu.edu.au:80/~jks/finder.html.

[SLM92] Bart Selman, Hector Levesque, and David Mitchell. A New Method for Solving Hard Satisfiability Problems. *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, San Jose, California, July 1992, pp. 440–446.

[SM96] Mark Saaltnik and Irwin Meisels. *The Z/Eves Reference Manual (draft)*. Technical Report TR-96-5493-03, ORA Canada, Ottawa, Canada, December 1995, revised April 1996.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*, Second edition, Prentice Hall, 1992.

[SRW99] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm, Parametric Shape Analysis via 3-Valued Logic. *Proceedings of ACM Symposium on Principles of Programming Languages (POPL '99)*,San Antonio, TX, Janunary 1999, pp. 105-118.

[SSX94] Norman Sadeh, Katia Sycara, and Yalin Xiong. *Backtracking Techniques for the Job Shop Scheduling Constraing Satisfaction Problem*. Technical Report CMU-RI-TR-94-31, The Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, 1994.

[Swi58] J. D. Swift. Isomorph Rejection in Exhaustive Search Techniques, in *Proceedings of 10th Symposium in Applied Mathematics of the American Mathematical Society*, April 1958, pp. 195–200.

[VHe89] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge, MA, 1989.

[Wal58] R. J. Walker. An Enumerative Technique for a Class of Combinatorial Problems. *Proceedings of 10th Symposium in Applied Mathematics of the American Mathematical Society*, April 1958, pp. 91–94.

[Wal75] David L. Waltz, Understanding Line Drawings of Scenes with Shadows. In *The Psychology of Computer Vision*, ed. P. H. Winston, McGraw Hill, Cambridge, MA, 1975, pages 19–91.

[Zha96] Jian Zhang. Constructing Finite Algebras with Falcon. *Journal of Automated Reasoning*, Vol. 17, No. 1, August 1996, pp. 1–22.

[ZS94] Hantao Zhang and Mark E. Stickel. *Implementing Davis-Putnam Algorithm by Tries*. Technical Report, The University of Iowa, 1994.

[ZZ95a] Jian Zhang and Hantao Zhang. Constraint Propagation in Model Generation. *Proceedings Principles and Practice of Constraint Programming - CP'95*, Cassis, France, Lecture Notes in Computer Science, Vol. 976, Springer-Verlag, September 1995, pp. 398–414.

[ZZ95b] Jian Zhang and Hantao Zhang. SEM: a System for Enumerating Models. *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI95)* Montreal, August 1995, Vol. 1, pp 298–303.

[ZZ96] Jian Zhang and Hantao Zhang. Combining Local Search and Backtracking Techniques for Constraint Satisfaction. *Proceedings of National Conference on Artificial Intelligence (AAAI-96)*, Vol 1, pp. 369–374.

[ZZ96b] Jian Zhang and Hantao Zhang. Generating Models by SEM. *Proceedings of International Conference on Automated Deduction (CADE-96)*, Lecture Notes in Artificial Intelligence 1104, Springer-Verlag, August 1996, pp. 308–312.